PREREQUISITES:

Euler tour of a tree, segment trees with lazy propagation, binary lifting

PROBLEM:

Given a tree on N vertices each with a color assigned to it, support Q queries on it, each being one of the following:

- Given u, change the root to u
- Given u and c, set every vertex in the subtree of u to have color c
- Given u and c, count the number of vertices in the subtree of u that have color c

EXPLANATION:

First, let's ignore the root-change updates and try to just solve the other two queries for a fixed root.

Dealing with color changes and queries

For now let's assume the root of the tree is 1.

A useful technique when dealing with updates and queries on subtrees is the Euler tour of a tree.
This allows us to convert subtree updates/queries into subarray updates/queries instead, which are much easier to deal with.

So, build the Euler tour of the given tree. Let B denote the array of colors in Euler tour order. We need to support the following:

- Given c and a range $[L, R]$, set $B_i := c$ for every $L \le i \le R$
- Given c and a range $[L, R]$, count the number of $L \le i \le R$ such that $B_i = c$

There are a few different ways to handle this. One is to use a segment tree with lazy propagation.

Details

Build a segment tree on B.
Each node of the segment tree stores a frequency map of all the elements in the subarray it represents. Let $freq[node]$ denote this map for a given node.

With this, querying a range for the count of c is trivial: it's simply the sum of $freq[node][c]$ across all the $\mathrm{O}(\log N)$ segtree nodes that cover the range $[L, R]$, so each query can be answered in $\mathrm{O}(\log^2 N)$ or such.

Handling updates can be done with the help of lazy propagation.

Suppose we want to set the range $[L, R]$ to c.

Let $[l, r]$ be one range of the segment tree that fully lies within $[L, R]$, corresponding to the node $node$.
Then, do the following:

- For each $x \in freq[node]$ and each ancestor a of $node$, perform `freq[a][x] -= freq[node][x]`
- Set $freq[node]$ to contain the single element c with a frequency of $r - l + 1$, and also increase $freq[a][c]$ by $r - l + 1$ for every ancestor
- Set a flag at this node to lazily propagate this information to its children when it is needed.

While this might make a single update take a lot of time, the total number of operations is bounded by $\mathrm{O}((N + Q)\log^2 N)$ map operations across all updates.

Proof

Each element of some map affects only its $\mathrm{O}(\log N)$ ancestors, and whenever we happen to process it, it is completely deleted.

Building the segment tree itself creates $\mathrm{O}(N \log N)$ map elements, and then each update creates $\mathrm{O}(\log N)$ more.
So, we create $\mathrm{O}((N+Q)\log N)$ map elements in total, each of which costs $\mathrm{O}(\log N)$ map operations to delete.

Note that depending on your implementation, you might need to use a reasonably fast hashmap to not get TLE: for example, the one **described here**.

Handling root changes

Changing the root appears to change the structure of the Euler tour completely, and we can't afford to rebuild it after each change.

However, as it turns out, we don't need to rebuild the Euler tour!

Let's root the tree at 1 and compute its Euler tour, as above. Now, consider two vertices r and u.

- If r does not lie in the subtree of u, then when rooting the tree at r, the subtree of u is the exact same as the current subtree of u.
- If r does lie in the subtree of u, then when rooting the tree at r, the subtree of u is the entire tree except the subtree of v, where v is the child of u that contains r.
  - Finding v is a simple application of binary lifting.

In terms of the Euler tour we first computed,

- The first case just corresponds to a single subarray in it
- The second case corresponds to the entire array except some subarray, which can be thought of as two disjoint subarrays instead

Either way, even if the root is changed we only need to update/query at most two ranges each time, so our time complexity doesn't change!

TIME COMPLEXITY:

$\mathrm{O}((N+Q)\log^2 N)$ or $\mathrm{O}((N+Q)\log^3 N)$ per testcase.

PROBLEM:

Given $1 \le X \le 10^{12}$, find whether there exist integers $1 \le A, B, C \le 10^6$ such that $AB + C = X$.

EXPLANATION:

If $X = 1$, no solution exists because $AB + C \ge 2$ for any choice of A, B, C.
Otherwise, a solution always exists!
There are many different approaches, I'll elaborate on one of them.

Notice that $X = AB + C$ is vaguely reminiscent of division.
In particular, if we fix the value of B, **Euclid's division lemma** tells us that there exist unique integers q (the quotient) and r (the remainder) such that:

- $X = qB + r$
- $0 \le r < B$

If we're able to choose B appropriately such that $1 \le q, r \le 10^6$, we'll be done, since we can directly take $A = q$ and $C = r$.
In fact, we don't really need to worry much about r. Since $r < B$, it'll never exceed 10^6 anyway, as long as we choose $B \le 10^6$.
So, our aim is to keep q small enough.

This can be achieved by picking a large enough B.
For example, suppose we pick $B = 10^6$ and then find q and r.
Then, since $X \le 10^{12}$, we'll surely have $q \le 10^6$, which takes care of all the upper bounds.

This solves the problem for almost all cases — the only ones it doesn't solve are the ones for which $q = 0$ or $r = 0$.
Let's see how we can fix those.

- If $q = 0$, that would mean $X < 10^6$.
  Solving such a case is quite easy though, since we have several options.
  For example, you can choose $A = B = 1$ and $C = X - 1$, or $A = C = 1, B = X - 1$.
- If $r = 0$, that would mean X is a multiple of 10^6.
  In such a case, we can decrease q by 1 and set $r = 10^6$.
  This will fail only for $X = 10^6$, since decreasing q by 1 will make it 0.
  However, $X = 10^6$ can be solved using the previous case anyway, so there's no issue.

Putting it all together,

- If $X = 1$, no solution exists.
- If $X \le 10^6$, choose $A = B = 1, C = X - 1$ (or similar).
- If $X > 10^6$, find q and r, the quotient and remainder when X is divided by 10^6. Then,
  - If $r > 0$, choose $A = q, B = 10^6, C = r$
  - If $r = 0$, choose $A = q - 1, B = 10^6, C = 10^6$

TIME COMPLEXITY

$O(1)$ per test case.

PROBLEM:

There's an array A of length N. You know the following pieces of information about it:

- M constraints of the form (pos, val), stating that $A_{pos} = val$.
- K constraints of the form (u, v, d), stating that $|A_u - A_v| \leq d$

Find out whether a valid array A exists.

EXPLANATION:

First, notice that we can replace the second inequality, on absolute values, by a couple of plain inequalities: $A_u - A_v \leq d$ and $A_v - A_u \leq d$

Inequalities of this type can be nicely 'chained' together, because $(A_u - A_v) + (A_v - A_w) = (A_u - A_w)$.
So, if we know constraints on (A_u, A_v) and (A_v, A_w), we also implicitly have a constraint on (A_u, A_w).
Of course, this applies to longer chains as well.

This information can be nicely represented by constructing an appropriate directed weighted graph.

In particular, consider a directed graph on vertices $1, 2, \ldots, N+1$, with edges defined as follows:

- If $|A_u - A_v| \leq d$ is known, add the edges $u \to v$ and $v \to u$, both with weight d.

This graph has a rather nice property: for any two vertices u and v, if there is a $u \to v$ walk with weight D, then $A_u - A_v \leq D$ will hold.
That is, the 'chaining' of inequalities is represented by walks in this graph.

Now, we also have to bring in the $A_{pos} = val$ information here.
That can be done by treating this as the pair of inequalities $A_{pos} \geq val$ and $A_{pos} \leq val$, or more specifically, $A_{pos} - 0 \leq val$ and $0 - A_{pos} \leq -val$.

So, let's add a new vertex 0 to the graph, whose value we'll pretend is always 0.
Now, if $A_{pos} = val$, add the edges $0 \to pos$ with weight $-val$ and $pos \to 0$ with weight val.

Now that we have all our constraints, we need to check if they're valid.

It can be seen that all the constraints are valid if and only if the graph doesn't contain negative cycles.

Proof

If the graph contains a negative cycle, then for any vertex i on this cycle we have the constraint $A_i < A_i$, which is obviously absurd.

Conversely, suppose the constraints aren't all valid, meaning there's always a conflict somewhere.
Then, there must exist some vertices u and v and a value d such that:

- Via some sequence of constraints, we know $A_u - A_v \leq d$
- Via some other sequence of constraints, we know $A_u - A_v > d$; which means $A_v - A_u < -d$

In particular, this gives us a $u \to v$ walk of weight $\leq d$, and a $v \to u$ walk of weight $< -d$

Joining these walks, we obtain a closed walk whose weight is negative.
Any such walk must contain a negative cycle as well, proving our claim.

Detecting whether a directed graph contains a cycle is a standard application of the Bellman-Ford algorithm, as seen **here** for example.

TIME COMPLEXITY

$\mathrm{O}(N \cdot (M + K))$ per testcase.

PROBLEM:

You have a binary array A. In one move, you can pick two indices $1 \le i < j < N$ such that $i + 1 < j$, and swap A_i with A_j and A_{i+1} with A_{j+1}.

Use at most N moves to sort A.

EXPLANATION:

Let c_0 and c_1 be the number of zeros and ones in A.

Let's assume $c_0 \le c_1$. In particular, this means $2c_0 \le N$.
For the array to be sorted, the first c_0 positions should be 0 and the last c_1 should be 1.

Now, we iterate i from 1 to c_0; at each step, we'll try to place a 0 at position i without affecting positions $1, 2, \ldots, i-1$; if we are able to do this for every $i \le c_0$, the array will be sorted.

This requires a little bit of casework.

- If $A_i = 0$ already, nothing needs to be done.
- Otherwise, let $x > i$ be the smallest index such that $A_x = 0$. We'll try to bring this 0 to position i.
  - If $x > i + 1$, we only need a single operation: (i, x).
  - If $x = i + 1$, we need two operations: first, perform the operation $(i + 1, i + 3)$ to move this zero away, then perform the operation $(i, i + 3)$ to bring it to position i.

We perform at most 2 operations for each 0, giving us a total of $\le 2c_0 \le N$ operations, as needed.
Notice that the $N \ge 7$ constraint is what allows our second case to work: we need $i + 4 \le N$, which will always be satisfied since $i \le c_0 \le \lfloor \frac{N}{2} \rfloor$.

Make sure to properly update the positions of the zeros when swaps are performed!

If $c_0 > c_1$, do the same thing but place 1's from the back instead.

There are other ways to solve this task, though most likely involving more casework and dealing with specific patterns.
One such example can be seen in the editorialist's code linked below, which only places 1's from right to left using ≤ 1 move for almost every position, but deals with multiple cases in the process.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Functional graphs

PROBLEM:

You have an array A with N elements from 0 to M, and a permutation P of length N.
For each i such that $A_i = 0$, you must replace it with some integer from 1 to M.

An array is beautiful if the following holds:

- For each i from 1 to N in order, replace A_i with A_{P_i}.
- If this eventually results in all the elements of A becoming equal, A is said to be beautiful.

Count the number of ways of replacing zeros such that the resulting array is beautiful.

EXPLANATION:

Let A be the initial array, and B be the array obtained by performing the given operation on A once.

Then, for each $1 \le i \le N$, there's a unique position pos_i such that $B_i = A_{\text{pos}_i}$
That is, the value ending up at position i has to come from some unique position, which depends only on what P is.

This is useful information to have, so let's see how we can compute it.

In particular, we can see that:

- If $P_i \ge i$, then $\text{pos}_i = P_i$, since position i just directly receives the value at position P_i
- If $P_i < i$, then $\text{pos}_i = \text{pos}_{P_i}$, because P_i receives its value from somewhere, then i receives this value from position P_i; so their sources are the same.

In this way, computing pos_i can be done in $\text{O}(N)$ for all i.

Now, let's attempt to use this information.
Consider a directed graph on N vertices, containing the N edges $i \to \text{pos}_i$ for each i.

Note that each vertex has exactly one outedge, so this is a functional graph.
In particular, we know what a functional graph looks like: several disjoint cycles, with trees hanging off of some vertices of the cycles (each tree is directed towards its corresponding cycle).

Analyzing this information, we can see the following:

- If a vertex is on a cycle, it will remain on that cycle forever. In particular, each step simply shuffles the values on a cycle within the vertices on it.
  - In particular, values on a cycle never disappear. So, if there are at least two distinct non-zero values on cycles, the answer is immediately 0.
  - If there is a non-zero element on a cycle, all zeros on cycles must be set to this value; so they are uniquely determined.
  - If all the elements on cycles are zeros, then there are M choices for what to choose for them.
- If a vertex is not on a cycle, it will eventually receive a value from a cycle; so its initial value doesn't matter at all.
  - If there are x indices such that $A_i = 0$ that are not on a cycle, they thus contribute M^x to the answer.

Together, this gives us a simple formula for the answer:

- If there are two distinct non-zero elements on cycles, the answer is 0.
- Otherwise, let there be x indices not lying on cycles such that $A_i = 0$ for these indices.
- If any cycle contains a non-zero element, the answer is M^x.

- Otherwise, the answer is M^{x+1}.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

You're given the order in which N programs are opened (or reopened).
Each time a program is (re)opened, it moves to the front of the list.

Find the final order of programs; print the last 2 characters of each one.

EXPLANATION:

Let S_i denote the i-th input string.

Instead of looking at the process in forward order, let's look at it in reverse order.

- Clearly, S_N is the last program opened, which means it'll be in the front of the list.
- S_{N-1} is the penultimate program opened.
  If it doesn't equal S_N, we know it's the second program in the list; otherwise it equals S_N and was already processed.
- Similarly, if S_{N-2} equals either S_N or S_{N-1}, then it would've been processed already.
  Otherwise, it's the next program in the list.
- More generally, for any $1 \le i \le N$,
  - If S_i equals one of $S_{i+1}, S_{i+2}, \ldots, S_N$, then it was already processed 'later'.
  - Otherwise, it's the next program in the list since everything in front of it was taken care of.

This gives us a straightforward algorithm:

- For each i from N down to 1, check if S_i equals one of $S_{i+1}, S_{i+2}, \ldots, S_N$, i.e, if it was seen before.
  To do this quickly, the simplest way is to maintain a set or map of strings, each time inserting S_i into it.
  This can be implemented with, for example, `std::set<string>` or `std::unordered_set<string>` in C++, `set` in Python, or `HashSet` in Java.
- If has occurred earlier, ignore it. Otherwise, print its last two characters.

TIME COMPLEXITY

$\mathrm{O}(N)$ set/map operations per testcase.

PROBLEM:

You're given a string S of length N. In one move, you can replace one of its characters with any other lowercase Latin letter.
Find the minimum number of moves required to make S an anti-palindrome, i.e, no rearrangement of S is a palindrome.

EXPLANATION:

First, let's check whether S is itself an anti-palindrome, in which case the answer will be 0.

To do this, we need to check whether S can be rearranged into a palindrome.
Checking for this is somewhat well-known, and can be done as follows:

- If N is even, then S can be rearranged to form a palindrome if and only if every character appears an even number of times.
- If N is odd, S can be rearranged to form a palindrome if and only if exactly one character appears an odd number of times; and everything else appears an even number of times.

Proof

Let's prove the even case first. It should be fairly obvious if you think about it a bit.

If S can be rearranged into a palindrome, then every element of S has an opposite 'pair' element in this rearrangement.
This means each character appears an even number of times in total, as claimed.

On the other hand, if every character appears an even number of times, we can then pair up copies of the same character and use them to form a palindrome by placing them at opposing sides of the string.

The odd case is similar: there's a unique 'middle' character which accounts for the odd frequency because everything else is paired up; but the proof is otherwise the same.

So, check if S satisfies the respective condition for its parity; if it doesn't, the answer is of course 0.

What about the case when S does satisfy those conditions, i.e, can be rearranged into a palindrome?

Let's analyze what we can change.

- If N is even, we know every character occurs an even number of times.
  just replace any character of S with any *other* character; and both the new and the old character will now have an odd frequency.
  So, the answer in this case is 1
- If N is odd, we need slightly more casework.
  Exactly one character has an odd frequency; let it be α. Then,
  - If S has some character that's *not* α, then one move is enough.
    Replace a non-α character with another different non-α character, now there are three characters with odd occurrences.
  - If S consists of *only* the letter α repeated N times, the answer is 2: replacing one occurrence of α with something else isn't enough (since the new character will have odd frequency but α will be even).
    However, after one replacement we'll be back in the first case; and from there we know the answer is 1.

Putting it together, we have the following cases:

- If S is already an anti-palindrome, the answer is 0.
- Otherwise,
  - If N is even, the answer is 1.
  - If N is odd and S contains at least two distinct characters, the answer is 1.

- Else, the answer is 2.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

You have M kilograms of crushed apples, and want to measure out exactly N of them.
To achieve this, you have a balance, which can only tell you whether the weights of both sides are equal or not.

Is it possible to measure out exactly N kilograms?

EXPLANATION:

First, if $N > M$ then obviously it's impossible to measure out N kilograms from M.

Now, let's try to figure out what can be measured out.

- If M is odd, there's nothing we can do: the balance can't be made equal, so we can't split the M kilograms any further.
- If M is even, then we can only get two batches of $\frac{M}{2}$ kilograms each.
  After this, we can recursively apply the same strategy to try and subdivide the $\frac{M}{2}$ kilograms further.

Putting these together, it's not hard to see that the parts we can get are of weights $M, \frac{M}{2}, \frac{M}{4}, \ldots, \frac{M}{2^k}$, where k is the largest integer such that 2^k divides M.
Going any further would be impossible since $\frac{M}{2^k}$ is odd.

Of course, we can also combine these parts together to form others: for example $\frac{M}{2} + \frac{M}{4} = \frac{3M}{4}$ can be formed.

Since $\frac{M}{2^k}$ is the smallest part we can form, and also divides all the other weights we form along the way, clearly any combination of weights we create will also be divisible by $\frac{M}{2^k}$.

So, it's necessary that N is a multiple of $\frac{M}{2^k}$, otherwise clearly measuring out N is impossible.
In fact, this condition is also sufficient!

Proof

Notice that we can form 2^k independent units of weight $\frac{M}{2^k}$, by simply repeatedly splitting any even weight into two smaller ones.

Then, we can simply combine $1, 2, 3, \ldots, 2^k$ of these units to form every multiple of $\frac{M}{2^k}$ that's $\leq M$, which is exactly what we wanted.

This gives us a pretty simple solution:

- Find k, the largest integer such that 2^k divides M.
  This can be done by just iterating over k starting from 0, since $M \leq 10^{18}$ ensures that $k \leq 60$.
- Then, the answer is `Yes` if $\frac{M}{2^k}$ divides N and `No` otherwise; of course along with $N \leq M$.

TIME COMPLEXITY

$\mathrm{O}(\log M)$ per test case.

PREREQUISITES:

Basic bit manipulation

PROBLEM:

You're given an array A and an integer Y.
Determine the smallest non-negative integer X such that $(A_1 \mid A_2 \mid \ldots \mid A_N \mid X) = Y$ or say that no such X exists.

EXPLANATION:

Let $Z = (A_1 \mid A_2 \mid \ldots \mid A_N)$. Computing Z is easy: just loop over the array.
We want to find X such that $(Z \mid X) = Y$.

Let's look at each bit individually. For a fixed bit b,

- If b is set in Y, then at least one of X and Z must have b set.
  - In particular, if Z has b set then we can leave it unset in X, since our aim is to minimize X.
  - On the other hand, if Z doesn't have b set, X *must* have it set.
- If b is not set in Y, then neither X nor Z can have it set.
  - In particular, if Z has b set then no valid X can exist, and the answer is immediately -1.

Repeating this process for each bit b tells us whether a valid X exists; and if it does, which bits to set in it to minimize its value.

While it's possible to implement this by looping over values of b from 0 to 20, there's a simpler way if you're familiar with bitwise operations:

- Note that a valid X can exist only when Z is a submask of Y. This can be checked in several ways, for example, `if ((Y & Z) == Z)`.
- When a valid X does exist, it contains exactly those bits that are set in Y but not Z, which is simply `Y ^ Z` since Z is a submask of Y (^ being xor here).

This gives us a very short implementation.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Computing GCD quickly

PROBLEM:

Find the maximum number of people who can split N apples and M oranges equally.

EXPLANATION:

Suppose there were k people. Then, for equal distribution, k must be a divisor of N *and* a divisor of M.

The largest such k is, by definition, $\gcd(N, M)$; and this is the answer.

Finally, since N and M can be quite large, it's necessary to compute $\gcd(N, M)$ quickly. This can be done with the help of the **Euclidean algorithm**, or you can just use inbuilt functions corresponding to your language (`std::gcd` in C++, `math.gcd` in Python, and so on).

TIME COMPLEXITY:

$\mathrm{O}(\log \min(a, b))$ per testcase.

PREREQUISITES:

Binary lifting or heavy-light decomposition

PROBLEM:

Given a tree on N vertices where the i-th vertex has A_i written on it, answer Q queries of the following form:

- Given u and v, find the longest arithmetic progression on the simple path from u to v.
  Note that this arithmetic progression should itself be a path.

EXPLANATION:

This is pretty much a pure data structure problem. There are a couple of different ways to solve it, I'll explain one below.

A simpler version

First, let's solve a simpler version of the queries: instead of arbitrary u and v, let's assume u is an ancestor of v (when the tree is rooted at 1).

To solve this, let's transform the problem a bit:

Root the tree at 1 and let p_u denote the parent of u.
Write the value $A_u - A_{p_u}$ on the edge between u and p_u.
Then, notice that "longest arithmetic progression on path" is now asking for "longest contiguous set of equal values on path", which is a bit easier to deal with.

Several different data structures can solve this query: for example, binary lifting or heavy-light decomposition.

The devil is in the details here: you'll need to maintain several different quantities and merge them correctly when doing binary lifting/segtree merging.
For instance, the editorialist's code linked below uses HLD, and maintains the following quantities for each segment tree node that represents some path:

- The answer for the path, i.e, the longest set of equal values
- The length of the path
- The first edge value seen on this path, and how many of them form the prefix
- The last edge value seen on this path, and how many of them form the suffix
- Merging two adjacent nodes requires merging these values appropriately, which takes a bit of casework (or represent the whole state as a matrix and use matrix multiplication, which is probably a bit less typing)

Binary lifting will require you to maintain several similar quantities in your lifting table.

Either way, this allows us to solve for a single query in $\mathrm{O}(\log^2 N)$ time, given that u is an ancestor of v.

The original problem

Let's now deal with arbitrary (u, v) queries.

Let $L = lca(u, v)$. Finding L can be done in $\mathrm{O}(\log N)$ or $\mathrm{O}(1)$ in various ways.

First, let's apply our above solution independently to (L, u) and (L, v).
Now, note that the only thing we're missing is paths that pass through L and don't have it as one endpoint.

However, such paths can be considered by just merging the answers for (L, u) and (L, v) appropriately: for example, keep the (L, u) answer as it is, and negate the differences of the (L, v) answer; then merge them.

Our merge function is $\mathrm{O}(1)$, so each query is now answered in $\mathrm{O}(\log^2 N)$.

Once again, this problem's difficulty mainly lies in correctly working out the details: for example, if you're using HLD, you need to be careful about the order in which merges are done, since the merge operation is not associative. I recommend looking through the code below if you're stuck and can't debug.

TIME COMPLEXITY:

$\mathrm{O}((N+Q)\log^2 N)$ per testcase.

PROBLEM:

You're given an array A and a string S which contains only L and R.
For each element of the string in order:

- If the array has size 1, do nothing.
- Randomly split A into two non-empty subarrays, each split being equiprobable.
- If S_i is L, discard the right part; otherwise discard the left part.

Find the expected sum of the final array.

EXPLANATION:

Every move, we cut out either a prefix or a suffix of the remaining array.
That means in the end, we'll be left with some subarray of A.
In fact, at any stage of the process we'll have a subarray of A.

Let $p(L, R)$ be the probability that we're left with subarray $[A_L, A_{L+1}, \ldots, A_R]$ after all the moves.
Then, by definition of expectation, the final answer is just $\sum P(L, R) \cdot (A_L + A_{L+1} + \ldots + A_R)$ across all $1 \leq L \leq R \leq N$.

So, it's enough for us to find all the $p(L, R)$ values.

The remaining subarray changes after each move, so we also need information about which move we're on.
Let $p(L, R, k)$ be the probability that after k moves, we have subarray $A[L : R]$ with us.
Initially, we have $p(1, N, 0) = 1$ and $p(L, R, 0) = 0$ for all other (L, R).

For $k \geq 1$, let's look at all possible moves.

- Suppose $S_k = \mathrm{L}$, meaning we must keep the left subarray.
  Then, subarray $[L, R]$ can be obtained by cutting $[L, x]$ for some $R < x$. So,

$$p(L, R, k) = \sum_{x=R+1}^{N} p(L, x, k-1) \cdot \frac{1}{x - L}$$

- If $L = R$, we'll also have an extra $p(L, L, k-1)$ term to account for size-1 subarrays not being splittable.
- If $S_k = \mathrm{R}$, transitions are similar: consider all subarrays $[x, R]$ for $1 \leq x < L$.

Notice that we have $\mathrm{O}(N^2 K)$ states, and $\mathrm{O}(N)$ transitions for each one.
Memoizing states thus allows for a $\mathrm{O}(N^3 K)$ dynamic programming solution, which is enough to pass subtask 1.

To optimize this solution further, it seems our only hope is to make transitions faster: all the states seem necessary.

One way to do this is to look at transitions from the opposite side.
Instead of looking at $p(L, R, k)$ in terms of sums of $p(L, x, k-1)$, let's look at how $p(L, R, k)$ *contributes* to $p(L, x, k+1)$

So, suppose $[L, R]$ is fixed, and $S_{k+1} = \mathrm{L}$, i.e, we keep the left part of the subarray.
There's a $\frac{1}{R-L}$ chance of obtaining each of $[L, L], [L, L+1], \ldots, [L, R-1]$.
So, we add $p(L, R, k) \cdot \frac{1}{R-L}$ to each of $p(L, L, k+1), p(L, L+1, k+1), \ldots, p(L, R-1, k+1)$.

Notice that L is fixed for all of these, so we're essentially just adding a constant value to a range!

Processing Q range-add updates on an array of length N can be done in $\mathrm{O}(N+Q)$ time using prefix sums.

How?

Suppose we want to process Q updates on the array B.

Create an auxiliary array C of range N, initially filled with zeros.
Then, for each update (L, R, x) (meaning we want to add x to the range $[L, R]$ of B),

- Increase C_L by x
- Decrease C_{R+1} by x

Finally, take the prefix sums of array C.
The i-th prefix sum of C is the amount by which B_i will change.

A slightly more detailed explanation of why this works can be found in the “How?” section **here** .

In our case we perform one update for each (L, R), for about N^2 updates in total.
So, for each move, we’re now down to $\mathrm{O}(N^2)$ operations in total!

This optimization makes the solution $\mathrm{O}(N^2K)$.

Note that once again, a little care is needed when dealing with $L = R$.

TIME COMPLEXITY

$\mathrm{O}(KN^2)$ per testcase.

PROBLEM:

Given a boolean array A of length N, in one move you can select two distinct indices i and j, and set both A_i and A_j to $A_i \oplus A_j$.

Find the minimum number of moves to obtain an array with an equal number of zeros and ones.

EXPLANATION:

If N is odd, the answer is clearly `-1`, so we only care about the even case.

Suppose the array initially has x zeros and y ones.
Note that since N is even, x and y will definitely have the same parity.

Our objective is to end up with $x = y$.
Let's analyze what exactly our operation can do:

- Operating on two zeros does nothing
- Operating on two ones changes them both to zeros. This increases x by 2 and decreases y by 2.
- Operating on a zero and a one changes the zero into a one, increasing y by 1 and decreasing x by 1.

With this in hand, we can do a bit of casework.

y ≤ x

If $x = y$ the answer is clearly zero.

Otherwise, we need to increase y and decrease x.
Notice that our operations only allow us to increase y by 1 and decrease x by 1 in one step.
In particular, this shortens the distance between x and y by two.

So, if $d = x - y$, we need exactly $\frac{d}{2}$ steps to make them equal.

However, there is one edge case here: if $y = 0$, then the answer is `-1`, since if the array is filled with zeros we can't actually turn anything into a 1.

y > x

Let $d = y - x$. We want to end up with $d = 0$.
Our operations allow us to either decrease d by 4 or increase it by 2.
So,

- If d is a multiple of 4, we need $\frac{d}{4}$ moves.
- If d is not a multiple of 4, we need $\frac{d}{4} + 2$ moves, since we need to spend one move increasing d by 2 before we reduce it to zero.

Once again, there's a small edge case here.
If d is not a multiple of 4 and $y = 2$, the answer is `-1`.
This is because the array will have two ones and no zeros, i.e, $A = [1, 1]$. There's no way to turn exactly one of these ones into a zero.

TIME COMPLEXITY

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Combinatorics or dynamic programming

PROBLEM:

Given an integer N, find the number of triplets X, Y, Z such that

- $0 < X, Y, Z < N$
- $X + Y + Z = X \mid Y \mid Z = N$

EXPLANATION:

For any integers X and Y, it's not hard to see that $X \mid Y \leq X + Y$, with equality holding if and only if X and Y don't share any bits in their binary representations.

This extends to three integers as well: $X \mid Y \mid Z \leq X + Y + Z$, and equality holds if and only if X, Y, Z all have distinct bits set.

Now, we know that $X \mid Y \mid Z = N$. This means that

- If a bit b is not set in N, then it can't be set in *any* of X, Y, Z
- If a bit b is set in N, then it should be set in at least one of X, Y, Z.
  Our earlier discussion further tells us that it can't be set in two or more of them, so it must be set in *exactly* one of them.

So, each set bit in N must be distributed to one of X, Y, or Z; while ensuring that each of them gets at least one set bit; our objective is to count the number of ways to do this.

There are several ways to do this, here are a couple.

Direct math

The number of ways can be calculated directly, using the inclusion-exclusion principle.

First, let's ignore the "each value gets at least one bit" condition, i.e, allow zeros.
Counting the number of ways here is simple: each set bit in N has exactly 3 choices, for whether it's given to X, Y, or Z.
So, if N has K set bits, the number of ways is 3^K.

Now, let's remove the cases when some of the values are 0.
If $X = 0$ in the end, all the bits were distributed to Y and Z, i.e, 2 options for each bit for 2^K in total.
By symmetry, this is also the number of ways in which $Y = 0$, or $Z = 0$.
So, the total number of ways here is $3 \cdot 2^K$.

Finally, we need to add in the number of ways where two or more of X, Y, Z are zero, since those would have been removed multiple times.
It's easy to see that there are only three ways here: $(N, 0, 0), (0, N, 0), (0, 0, N)$.

So, the answer is simply

$$3^K - 3 \cdot 2^K + 3$$

Dynamic programming

Suppose N has K set bits.
As our observations showed, only this number K matters; which bits were set doesn't matter at all.

Our aim is to find out the number of ways to split K bits into three non-empty subsets.
Since K is quite small ($N < 2^{60}$, so $K \leq 60$), this can be done using dynamic programming.

Let dp_i be the number of ways to split i bits into three non-empty subsets.

Then, if we fix the size of the first subset j $(1 \leq j < i)$, we have:

- $\binom{i}{j}$ ways to choose which bits go into this subset.
- $2^{i-j} - 2$ ways to distribute the remaining bits into two non-empty subsets.

So, we have

$$dp_i = \sum_{j=1}^{i-1} \binom{i}{j} (2^{i-j} - 2)$$

which can easily be precomputed for all $K \leq 60$ in $\mathrm{O}(K^2)$ or $\mathrm{O}(K^3)$, after which answering queries is easy.

TIME COMPLEXITY

$\mathrm{O}(\log N)$ per test case.

PREREQUISITES:

Binary search, finding centroids of a tree

PROBLEM:

You're given a tree on N vertices.
Two hidden vertices of this tree are marked as special, and the path between them is called the beautiful branch.
Your task is to find any one of the special nodes.

To achieve this, you can ask upto 300 queries of the following type:

- Provide a set of vertices $v_1, v_2, \ldots, v_k$ to the judge.
- The judge will return $\sum_{i=1}^{k} d(v_i)$, where $d(v_i)$ is the distance between v_i and the beautiful branch.

Also, the total sizes of subsets queried cannot exceed $35N$.

EXPLANATION:

Let's root the tree at vertex 1.

Now, suppose we query for the single vertex 1 and obtain $d(1)$.
Also, suppose it has k children, say $v_1, v_2, \ldots, v_k$.
We have the following cases:

- If 1 is the unique special vertex, we'll have $d(v_i) = 1$ for every v_i.
- If 1 is one end of the beautiful branch, we'll have $d(v_i) = 0$ for exactly one child, and $d(v_i) = 1$ for all the rest.
- if 1 lies on the beautiful branch but isn't an endpoint, we'll have $d(v_i) = 0$ for exactly two children and $d(v_i) = 1$ for the rest.
- The only remaining case is when the beautiful branch doesn't contain 1 at all.
  In this case, it'll be contained in the subtree of one of its children, say v_i.
  Then, we'll have $d(v_i) = d(1) - 1$, and $d(v_j) = d(1) + 1$ for all other v_j.

If we're able to quickly distinguish between these cases, we can figure out which subtree to move into and recursively solve for that subtree.

The fourth case is somewhat easy to deal with. We know it'll only happen when $d(1) > 0$ so it can be treated separately.
We want to find the unique child with a smaller value than the rest.
Let's binary search on the children from 1 to k, each time querying for the vertices $v_1, v_2, \ldots, v_i$.
If this sum equals $i \cdot (d(1) + 1)$, the child we're looking for is after i; otherwise it's between 1 and i.
This way, we take $\mathrm{O}(\log N)$ queries to find the appropriate child, and we can recurse.

Now we need to distinguish between the other three cases.
Case 1 is easy: it happens if and only if $\sum_{i=1}^{k} d(v_i) = k$, so it can be checked for with one query.

Cases 2 and 3 can in fact be combined, and once again solved with binary search.
We just want to find one child that the beautiful branch goes into, so we can for example binary search to find the smallest i such that $d(v_1) + d(v_2) + \ldots + d(v_i) < i$, since all values are either 0 or 1.

Once the appropriate child is found, we can recurse into it and once again use this procedure, till we find an endpoint (which refers to hitting case 1).

This works … almost.
We need to think about the recursion depth. We use upto $\log N$ queries each time, so if we recurse D times,

we'll use $\mathrm{O}(D \log N)$ queries in total. This number needs to be kept within 300.

Here, we make the observation that we don't quite need to move directly to a child of vertex 1: we move into its *subtree*, so we're free to choose our root within this subtree.

This freedom of choice of root is quite nice, since it allows us to bound the recursion depth nicely.
At each step, choose the *centroid* of the tree you're in.
That way, each time we move into a subtree we know its size at least halves, so after $\log N$ steps we'll reach a tree of size 1 which is trivial to solve.
This way, we use about $\log^2 N$ queries, which fits within 300.

However, doing this slightly breaks other parts of our solution!
In particular, cases $1, 2,$ and 4 from above remain the same.
However, case 3 requires care.

When there are two possible branches to move into, it's not always correct to move into an arbitrary one.
This is because when we moved into this subtree, it's possible it contained only one of the two special vertices (for example, if we entered it via cases 2 or 3).
So, while the beautiful branch may be present in both subtrees, only one of them will contain an endpoint of it, and we need to choose this carefully.

One way to resolve this issue is to use distances.
Suppose we solve for centroid c_1, and have decided which subtree to go into.
Let c_2 be this subtree.
If we run into an ambiguity at c_2, it means that the branch passes through c_2 — in particular, it means one endpoint is in the subtree of c_2 (when c_1 is considered as the root).

So, let's compute the distances of all vertices from c_1.
Then, if there happens to be an ambiguity at c_2, simply choose the child that's further away from c_1.

Note that this requires us to find both children of c_2 that might satisfy the condition instead of just one, hence needing two binary searches.
However, this still fits within the query limit.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PROBLEM:

Given an array A, find the largest integer X such that $A_i \in [-X, X]$ for *every* i.
If no such X exists, print -1.

EXPLANATION:

First, notice that if A contains a 0, the answer is -1. This is because $[-X, X]$ will *always* contain 0 for any $X \geq 0$.

Now, suppose a valid X existed. Looking at some A_i,

- If $A_i > 0$, then $X < A_i$ must hold.
- If $A_i < 0$, then $A_i < -X$ must hold. This is the same as saying $X < -A_i$.

So, X must be strictly less than *every* positive element of A, and also strictly less than the negative of *every* negative element of A.

This gives us a simple way to find the maximum possible X:

- Find the smallest positive A_i in the array; let this value be P.
- Find the largest negative A_i in the array; let this value be N.
- The largest valid X is then $\min(P, -N) - 1$.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Greedy algorithms

PROBLEM:

A string S is called good if, for every suffix T of S and characters c_1, c_2 that occur in S, the frequencies of c_1 in T and c_2 in T differ by at most K.

Given S, find its lexicographically smallest good rearrangement.

EXPLANATION:

Let's first figure out when exactly a string S has a good rearrangement.
We'll call a suffix that satisfies the given condition *balanced*.

Let c_1 and c_2 be some two characters that occur in S, and f_{c_1} and f_{c_2} be their respective frequencies.
If S is to be good, clearly $|f_{c_1} - f_{c_2}| \le K$ must hold; otherwise the string itself is not balanced, no matter how we rearrange it.

In fact, this condition is also sufficient for S to have a good rearrangement, i.e, $|f_{c_1} - f_{c_2}| \le K$ for every pair of characters c_1, c_2 in S.

Proof

We can construct a good rearrangement greedily.

Suppose we have N characters, such that $|f_{c_1} - f_{c_2}| \le K$ for every pair of them.
Construct a string as follows:

- Let c be a character with maximum frequency. Place one occurrence of c, and decrement f_c by 1.

It's easy to see that this preserves the condition $|f_{c_1} - f_{c_2}| \le K$, and so we can place all N characters.
Further, the string we construct is good, since at each step the suffix we create is balanced.

Notice that instead of checking every difference, it's enough to check if the maximum difference satisfies the condition.
That is, a string has a good rearrangement if and only if the difference between its maximum and minimum frequency characters is $\le K$.

Now that we know how to check whether S has a good rearrangement, we need to figure out how to construct the lexicographically smallest one.
Constructing lexicographically smallest objects usually involves doing so greedily, and this task is no different.

For each i from 1 to N, let's greedily try to place the characters 'a', 'b', 'c', $\ldots$, 'z' in order. As soon as one of them can be placed, do so and go to the next index.
To check whether placing a certain character is possible, we need to check if the suffix of the string starting from $i+1$ has a good rearrangement or not.

However, we have a pretty simple condition for this: the difference between the maximum and minimum frequencies of the remaining characters should be $\le K$.
We know which characters we've placed so far, so we also know the frequencies of the remaining characters.
So, we can check this condition in $\mathrm{O}(\Sigma)$ by just directly computing the maximum and minimum frequency, where Σ is the alphabet size (here, $\Sigma = 26$).

For each position, we try (at most) 26 possibilities, and process each in $\mathrm{O}(26)$.
So, our solution runs in $\mathrm{O}(26^2 \cdot N)$, which is fast enough.
It's possible but unnecessary to make it run in $\mathrm{O}(26 \cdot N)$ with some minor optimizations.

TIME COMPLEXITY

$\mathrm{O}(26 \cdot N)$ per test case.

PROBLEM:

The beauty of an array equals the sum of pairwise products of its elements.
You're given an array A. At most K times, you can increase one of its elements by 1.
What's the maximum possible beauty of the final array?

EXPLANATION:

There are three parts to this problem:

1. Computing the beauty of an array faster than $\mathrm{O}(N^2)$.
2. Figuring out which elements to apply the operations to.
3. Actually applying these operations quickly, since $\mathrm{O}(K)$ would be too slow.

Let's go over these one at a time.

Computing beauty

We want to quickly compute the sum of pairwise products of elements.
Let $S = \sum_{i=1}^{N} \sum_{j=1}^{N} (A_i \cdot A_j)$ be this quantity. Then,
Note that

$$\begin{aligned}(A_1 + A_2 + \ldots + A_N)^2 &= (A_1^2 + A_2^2 + \ldots + A_N^2) + 2 \cdot \sum_{1 \le i < j \le N} A_i \cdot A_j \\ &= (A_1^2 + \ldots + A_N^2) + 2S\end{aligned}$$

So, S can easily be computed in $\mathrm{O}(N)$: we only need to know the square of the sum of the elements, and the sum of squares of the elements.

Now, let's look at what happens when we need to perform operations.
Suppose we had $K = 1$, i.e, only one operation needed to be performed. What would the optimal move be?
From the way we rewrote the cost above, it's obvious that we should choose the smallest A_i to increase by 1.

In fact, this greedy choice holds even for $K > 1$: for each move, choose the smallest element of A and increase it by one!

Proof

This will be a bit long to write out, though it's really just several applications of exchange arguments.

Let $A_1 \le A_2 \le \ldots \le A_N$.
Suppose we increase A_i exactly B_i times; so the final values are $C_i = A_i + B_i$.

Recall that the beauty is (half of) the square of the sum of C_i, minus the sum of squares of C_i.
Note that the sum of C_i is always a constant (and equals K plus the sum of A_i).
So, maximizing the beauty requires us to minimize the sum of the squares of C_i.

Let's call a *solution* an assignment of B_i values, and an *optimal solution* a solution that maximizes beauty.

Claim 1: There exists an optimal solution such that $C_i \le C_j$ when $i \le j$.
Proof: Trivial, if $C_i > C_j$ then since $A_i \le A_j$ it's always possible to shift operations from index i to index j and essentially swap C_i and C_j.

From now on, we'll only consider optimal solutions with this structure.

Claim 2**:** In an optimal solution, if $i < j$ and $B_j > 0$, then $B_i > 0$.
Proof: If $B_i = 0$, reduce B_j by 1 and increase B_i by 1.
Work out the algebra to see that this gives us a strictly better beauty, which contradicts us starting with an optimal solution.

This tells us that we'll only operate on some *prefix* of indices.
Let m be the highest index operated on.

Claim 3**:** If $1 \le i < j \le m$, then $C_j - C_i \le 1$
Proof: If C_i and C_j differ by two (or more), reduce one operation from the larger one and give it to the lower one; this improves beauty, which is once again a contradiction.

Note that these properties (almost) uniquely characterize an optimal solution.
In particular, there'll be an index m such that:

- The suffix starting from A_{m+1} is unchanged, i.e, $C_i = A_i$ for this suffix.
- There'll be some integer x such that every C_i upto m is either x or $x + 1$.

Note that if m is fixed, x is also uniquely fixed since we must perform exactly K moves.
Further, m itself is essentially fixed: we need to (at least) bring all the elements up to A_m, requiring $(A_m - A_1) + (A_m - A_2) + \ldots + (A_m - A_m)$ moves.
This is an increasing function of m and we want it to stay within K, so the breakpoint is essentially unique.

Finally, note that the greedy solution of "always increase the smallest element" achieves this exact structure as well; and hence is optimal.

Now, let's see how we can use this information to solve the problem at hand.

Let's sort A, so that $A_1 \le A_2 \le A_3 \le \ldots \le A_N$.
Now, look at the process of performing operations:

- First, we increase only A_1 till it reaches A_2.
- Then, we alternate increasing A_1 and A_2 till they reach A_3.
- Then, we A_1, A_2, A_3 in turn till they all reach A_4.

  $\vdots$
- We increase $A_1, \ldots, A_{i-1}$ in turn till they all reach A_i.

This allows us to quickly simulate the process by combining operations.
Let's iterate from 1 to N.
When at index i:

- We've ensured that so far, $A_1 = A_2 = \ldots = A_i$.
  Note that they'll all be equal to (the initial value of) A_i.
- Next, we must perform operations till all these elements reach A_{i+1}.
  This takes exactly $i \cdot (A_{i+1} - A_i)$ operations.
- If K is at least this number, we know that all the operations can be performed, so we move to index $i + 1$.
- Otherwise, the process stops at this index.
  Since the operations are to be performed on the elements in turn, the exact values of all of $A_1, \ldots, A_i$ can be figured out; and $A_{i+1}, A_{i+2}, \ldots, A_N$ won't change.

We do $\mathrm{O}(1)$ work at each index; and $\mathrm{O}(N)$ work at one index, so this is overall linear.
Since we required sorting, the complexity is $\mathrm{O}(N \log N)$.

Alternately, binary search on the value of the minimum of the array for a solution in $\mathrm{O}(N \log K)$.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

Centroid decomposition, point-update range-query data structures

PROBLEM:

You're given a tree on N vertices, vertex i has the value A_i.
For two vertices x and y, let $f(x,y)$ denote the minimum value on the path between them, and $g(x,y)$ denote the maximum.
Find $\sum_{x<y} f(x,y) \cdot g(x,y)$.

EXPLANATION:

First, let's try to solve easier versions of this problem.

For a fixed root vertex r, computing $\sum_{x=1}^{N} f(r,x) \cdot g(r,x)$ can be done easily in $\mathrm{O}(N)$ time using a simple DFS to compute the required values.
However, there doesn't seem to be a nice way to quickly recompute this value when changing roots.

Instead, we'll try to do a bit more: we'll try to compute the sum of $f(x,y) \cdot g(x,y)$ for all pairs of vertices (x,y) whose paths *pass through* r. Let's see how we can do this quickly.

First, compute $f(r,x)$ and $g(r,x)$ for all vertices; as mentioned above, this can be done in $\mathrm{O}(N)$ using a DFS.
For convenience, let $B_x = f(r,x)$ and $C_x = g(r,x)$.

Notice that for a path (x,y) that passes through r, it holds that $f(x,y) = \min(B_x, B_y)$ and $g(x,y) = \max(C_x, C_y)$.
So, we essentially want to compute $\sum \min(B_x, B_y) \cdot \max(C_x, C_y)$ across all valid pairs (x,y).

Once again, let's relax the problem a bit, ignore the "valid pairs (x,y)" condition, and just attempt to quickly compute that summation for *all* pairs.

How?

Let's sort the pairs of (B_x, C_x), so that $B_1 \le B_2 \le \ldots B_N$.

Then, for each i from 1 to N:

- Let's only consider $i < j \le N$
- The sorting guarantees that $\min(B_i, B_j) = B_i$, so we only need to deal with C_j values.
- There are two cases here: $C_i \ge C_j$ and $C_i < C_j$.
  - If $\max(C_i, C_j) = C_i$, we add $B_i \cdot C_i$ to the answer. Overall, the answer increases by $k \cdot B_i \cdot C_i$, where k is the number of indices such that $C_i \ge C_j$.
    Processing them all quickly can be done if we can just find k.
  - If $C_i < C_j$, we add $B_i \cdot C_j$ to the answer. The answer increases by $B_i \cdot S$, where S is the sum of all C_j that are larger than C_i.
    Once again, we can process them all quickly if we can quickly compute S.

So, we need to be able to quickly answer the following questions about the suffix from $i+1$:

- How elements are $\le C_i$?
- What's the sum of elements that are $> C_i$?

If i is iterated in decreasing order from N down to 1, both of these can be answered in $\mathrm{O}(\log N)$ by a fenwick/segment tree built on values.

This allows us to compute the entire sum in $\mathrm{O}(N \log N)$.

Now that we've done this, we have some excess that needs to be removed.
In particular, we need to remove the sum of $\min(B_x, B_y) \cdot \max(B_x, B_y)$ for all pairs vertices whose paths don't pass through r.

In fact, such pairs can be found quite easily.
Let's root the tree at r, and let $c_1, c_2, \ldots, c_m$ be its children.

What we need to remove is then the value of all pairs that lie in the subtree of c_1, all pairs in the subtree of c_2, $\ldots$, all pairs in the subtree of c_m.
This is quite easy to do: after all, we already have a way to compute the sum of all pairs of vertices, given the list of their values. So, simply run the above on each of the respective sets, and subtract it from the answer.
This also takes $\mathrm{O}(N \log N)$ time in total, since the sum of the sizes of the sets is $N - 1$.

Now that we've done this, we can move on to a solution to the original problem.

Notice that if we root the tree at some r and compute this value, we are left with several smaller trees (one corresponding to each neighbor of r); which can be solved with the same algorithm.

It's well-known from the centroid decomposition algorithm that choosing r to be the centroid each time leads to a depth of $\mathrm{O}(\log N)$, and at each level of this recursion we do a total of $\mathrm{O}(N \log N)$ work leading to a solution in $\mathrm{O}(N \log^2 N)$ overall.

TIME COMPLEXITY

$\mathrm{O}(N \log^2 N)$ per test case.

PREREQUISITES:

Bitmasks

PROBLEM:

Given N, find the minimum number of distinct powers of 2 or 3 that sum to N.

EXPLANATION:

First off, notice that there's almost no intersection between the powers of 2 and 3: the only common power they share is 1.
However, it's never going to be optimal to take both 2^0 and 3^0 so we can ignore this intersection entirely. Do you see why?

In particular, this means the distinct condition is taken care of nicely for us, so we only need to decide on the set of powers of 2 and 3 that we take.

If we didn't have the powers of three to deal with, this would be a pretty simple problem: the minimum number of powers of 2 required to represent a number equals the number of ones in its binary representation.

Notice that N is pretty small.
In particular, since $N \leq 10^8$, we have $3^{17} > 10^8 \geq N$ for any input N.

This means the only powers of three we care about are $\{3^0, 3^1, 3^2, \ldots, 3^{16}\}$.

So, let's fix which subset of these powers we're taking into the sum.
There are 2^{17} such subsets.

Let S be the sum of the chosen subset.
We now need to write $N - S$ as the sum of powers of two, which we already know how to do: we just need to count the number of ones in the binary representation of $(N - S)$ which is easy.

This solution can be further optimized a bit by not checking all 2^{17} subsets for every testcase, and instead only checking the subsets of $\{3^0, 3^1, \ldots, 3^{\lfloor \log_3 N \rfloor}\}$.
This is faster because the sum of N is bounded by 10^8.

TIME COMPLEXITY

$\mathrm{O}(2^B \cdot B)$ per testcase, where $B = \lfloor \log_3 N \rfloor$.

PROBLEM:

Given an integer N, find four distinct positive integers a, b, c, d such that $(((a \ \& \ b) \mid c) \oplus d) = N$.

EXPLANATION:

There are several ways to solve this problem. Here are a few:

Author's solution (fix c)

Let's fix $c = 2^{32} - 1$, i.e, the first 32 bits of c are set.

This will make $(a \ \& \ b) \mid c)$ always equal $2^{32} - 1$, no matter what a and b are.

So, choose $d = c \oplus N$, and then arbitrary a and b that don't equal c or d, and we're done.

There are only a couple of edge cases here: when we get $d = 0$ or $d = c$.

- If $d = 0$, that means $N = 2^{32} - 1$. Find any solution for this case by hand and print it separately.
- If $d = c$, that means $N = 0$. Again, find a solution to this case by hand.

There are other ways to do this too; for example, you can fix $c = 1$ and similarly solve all but a handful of cases.

Tester's solution (brute force lower bits)

Let's solve the problem for $N < 8$ by brute force, while ensuring that a, b, c, d are all also < 8.
Simply brute-forcing all 8^4 possibilities here will tell you that this is possible.

To solve for $N \geq 8$,

- Set the first three bits of a, b, c, d to the solution for $N \mod 8$.
- Give all higher bits that are set in N to d.

Editorialist's solution (Random)

Choose a, b, c randomly between 1 and 10^{18}, and then choose $d = N \oplus ((a \ \& \ b) \mid c)$.

TIME COMPLEXITY

$\mathrm{O}(1)$ per test case.

PREREQUISITES:

Sweep line

PROBLEM:

Given an array of length N, initially filled with zeros, process Q updates on it of the following form:
Given an index i and a range x,

- For each $0 \le j \le x$, add $j \cdot (j+1)$ to A_{i+j}
- For each $0 \le j \le x$, subtract $j \cdot (j+1)$ from A_{i-j}

Print the final array after all updates are done.

EXPLANATION:

Since we only need to find the final array after all operations are performed, the queries can be processed offline.

Notice that the addition/subtraction updates are symmetric, so let's see how to handle only the addition updates; the subtractions can be handled similarly by reversing the array.

So, let's look at a single update (i, x). We add $j \cdot (j+1)$ to index A_{i+j}.
Let's expand this into $j^2 + j$, which we'll deal with separately.
As a result, we need to be able to:

- Add $1, 2, 3, \ldots$ to a range.
- Add $1^2, 2^2, 3^3, \ldots$ to a range.

Of course, this is doable using a segment tree; but below I'll present a simpler solution without data structures.

First, let's look at only adding $1, 2, 3, \ldots, R - L + 1$ to the range $[L, R]$.
Let's mark this update as starting at position L and ending at position R.

After doing this marking for every query, let's iterate over positions of the array from 1 to N and analyze what happens.
Suppose we've computed the values of $A_1, A_2, \ldots, A_i$ and now want to compute A_{i+1}.
When moving from index i to index $i+1$, we have the following changes:

- Every update that contributed a value of y to A_i will contribute a value of $y+1$ to A_{i+1}, since we're moving one step right.
  - That is, if A_i was affected by k distinct updates, i.e, $A_i = y_1 + y_2 + \ldots + y_k$, then these updates will contribute a value of $(y_1 + 1) + (y_2 + 2) + \ldots + (y_k + 1)$, which simply equals $A_i + k$, to A_{i+1}.
  - Notice that this change can be calculated in $\mathrm{O}(1)$ if we know k, the number of 'active' updates. This can be easily maintained in a variable as we iterate.
- Second, every update that *starts* at index $i+1$ will contribute 1 to A_{i+1}.
  - For each update that starts at index $i+1$, add 1 to A_{i+1}.
  - Recall that we were also maintaining the number of 'active' updates, k. So, for each such new update, increase k by 1.
- Finally, every update that *ends* at index i should no longer be counted in the summation.
  - For each update, we've already marked where it ends. When doing this, we can also store the final value of this update, at which point this final value can be directly subtracted from A_{i+1}.
  - Once again, remember to subtract 1 from k so that the number of active updates is maintained correctly.

In this way, we can move from index i to index $i+1$ somewhat quickly.
Notice that each update is processed exactly twice: once when it starts, and once when it ends.
Apart from that, we do $\mathrm{O}(1)$ work at each index, so the total complexity is just $\mathrm{O}(N+Q)$.

Now, we've processed all updates of the form "add $1, 2, 3, \ldots, R-L+1$ to $[L, R]$".
We still need to process the updates of the form "add $1^2, 2^2, 3^2, \ldots, (R-L+1)^2$ to $[L, R]$".

In fact, that can be handled in almost the same way!

Recall that we had three separate parts when moving from index i to $(i+1)$ — calculating the updated contribution of existing updates, dealing with updates that start at index $i+1$, and dealing with updates that ended at index i.

The second and third parts here are exactly the same: starting a new update adds 1 to the value of A_{i+1}, and when an update ends we've already stored its ending value, so just subtract its square from A_{i+1}.

The only difference here is the first part, i.e, updating the contribution of existing updates.
In particular, we have $y_1^2 + y_2^2 + \ldots + y_k^2$, and we want to compute $(y_1+1)^2 + (y_2+1)^2 + \ldots + (y_k+1)^2$.

This can be done by expanding out that summation to obtain

$$\begin{aligned}\sum_{i=1}^{k} (y_i+1)^2 &= \sum_{i=1}^{k} (y_i^2 + 2y_i + 1) \\ &= \sum_{i=1}^{k} y_i^2 + 2\sum_{i=1}^{k} y_i + k\end{aligned}$$

Notice that we already know k and $\sum_{i=1}^{k} y_i^2$.
We also know $\sum_{i=1}^{k} y_i$, since that's the value we were maintaining for the first part anyway!

So, once again all Q of the sum of squares updates can be processed in $\mathrm{O}(N+Q)$ time.

Finally, do the same for the subtraction updates, and we're done.

TIME COMPLEXITY

$\mathrm{O}(N+Q)$ per test case.

PROBLEM:

A string is called *boring* if all its characters are the same.
Given a string S, find the longest boring substring of S that appears more than once.

EXPLANATION:

First, note that boring substrings corresponding to different characters are independent.
So, let's fix a character, say c, and compute the longest boring substring consisting of c that appears more than once. We can then repeat this for every letter and take the maximum across all answers.

Solving this problem hinges on a single major observation:
Suppose S has a boring substring of length k. S then contains **two** boring substrings of length $k-1$: the first $k-1$ and the last $k-1$ characters of the substring.

So, let's fix a character c and compute the longest boring substring of S consisting of c.
This is fairly easy to do in $\mathrm{O}(N)$ by just iterating across the string and maintaining the current length of the boring substring: increase it by 1 when you encounter a c and reset it to zero otherwise.

Let this length we compute be k. Then,

- If there are two separate boring substrings of length k, the answer for c is k
- Otherwise, the answer for c is $k-1$.

All that remains is to check the first condition. This is fairly easy to do: for example, when computing k, keep a second variable that counts the number of occurrences of k and update it each time you update k.

The above approach is $\mathrm{O}(26N)$, but it's not hard (though also unnecessary) to optimize it to $\mathrm{O}(N)$ by solving for all characters simultaneously.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

You're given an array A of length N.
It's known that $2 \cdot (A_1 \oplus A_2 \oplus \ldots \oplus A_N) \geq (A_1 \mid A_2 \mid \ldots \mid A_N)$.

Do the following N times:

- Choose an element x remaining in the array.
- Either create a new box and add x to it, or add x to an existing box whose XOR is at least x.

Find the minimum number of boxes needed.

EXPLANATION:

Let's make some observations.

Let B be the maximum bit present in some array element.
Let's call some A_i *good* if it contains B.

Then, note that:

- No box can contain ≥ 3 good elements.
- If a non-empty box contains only non-good elements, we can never insert a good element into it.
- If a box contains exactly one good element, we can insert as many non-good elements into it as we like.

Proof

All three claims are fairly easy to prove once you've written them down.

- When the second good element is inserted into a box, the xor of this box no longer contains bit B; which means it only contains bits strictly smaller than B.
  This means any good element is strictly larger than this XOR, and can no longer be inserted into it.
- The second claim also follows from similar logic: with only non-good elements, any good element is strictly larger than their XOR since only bits less than B are used.
- The third claim is similar: with exactly one good element, the XOR will contain B.
  This makes it strictly larger than any non-good element, so any such element can always be inserted.

So, suppose there are k good elements in the array.
Then, notice that we surely need at least $\lceil \frac{k}{2} \rceil$ boxes; since each box can contain at most 2 good elements.

If k were odd, using exactly $\lceil \frac{k}{2} \rceil$ boxes is possible, as follows:

- Keep one good element aside, and pair the rest.
- Put the unpaired element into one box, and then all the non-good elements.
- Then, for each pair of good elements, put the larger one into a new box, then the smaller one.

Now, recall the condition $2 \cdot (A_1 \oplus A_2 \oplus \ldots \oplus A_N) \geq (A_1 \mid A_2 \mid \ldots \mid A_N)$.

This really just means that B, the largest bit, is present in the XOR of all elements.
This is only possible when B occurs an odd number of times, i.e, the number of *good* elements is odd.

We've already solved the odd case, so we're done!

TIME COMPLEXITY

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Combinatorics

PROBLEM:

A binary string is said to be good if it has an equal number of 01 and 10 substrings.

Given a binary string S, count the number of its good subsequences.

EXPLANATION:

First, we need a simpler characterization as to exactly when a binary string is good.

Let A be a binary string.
Suppose its first character is 0.
Then, A looks like a bunch of zeros, followed by several ones, followed by several zeros, and so on.
Since we only care about adjacent characters when they aren't equal, we can compress the string, i.e, $A = 010101\ldots$.

In this form, it's not hard to see that A has an equal number of 10 and 01 substrings only when it ends with a 0.
This condition is both necessary and sufficient; more importantly, it's quite a simple one, since it depends only on the first and last characters.

This is the characterization we'll use: a binary string A is *good* **if and only if** $A_1 = A_N$.

Given S, our aim is now to count the number of subsequences of S that start and end with the same character.
First off, there are N subsequences of length 1 that always fit this criterion, so let's keep them aside.

Suppose we have $S_i = S_j$, where $i < j$.
Let's count the number of subsequences with these two as the endpoints.

- We can't pick any elements before i or after j.
- Between i and j, we can freely pick any elements.
- There are $(j - i - 1)$ elements between them, so there are 2^{j-i-1} subsequences in total.

This gives us a (slow) solution in $\mathrm{O}(N^2)$ by fixing each pair of (i,j). All that remains is to optimize it.

Let's fix the value of j.
Then, we want to compute the sum of 2^{j-i-1} across all $i < j$ such that $S_i = S_j$.
Notice that $2^{j-i-1} = 2^{j-1} \cdot (2^{-i})$, so all we really want is the sum of 2^{-i} across all $i < j$ such that $S_i = S_j$: we can later multiply this sum by 2^{j-1} to account for *all subsequences ending at index* j.

Maintaining this sum is quite easy!
Computing 2^{-i} for a given i is a standard application of binary exponentiation and modular inverses.
After that, the only thing that needs to be maintained is two separate sums: the sum of 2^{-i} for all indices i such that $S_i = 0$, and the same sum for all indices such that $S_i = 1$.

Then, for a fixed j, the required contribution can be computed in $\mathrm{O}(1)$; after which the corresponding sum can be increased by 2^{-j}.

TIME COMPLEXITY

$\mathrm{O}(N)$ or $\mathrm{O}(N \log MOD)$ per testcase.

PREREQUISITES:

[Pigeonhole principle]

PROBLEM:

Given a weighted tree on N vertices, find two distinct paths whose bitwise XOR is the same; or claim that no such pairs exist.

EXPLANATION:

The A_i values are between 0 and 2^{20}, which means that the bitwise XOR of any path is at most $2^{21} - 1$, i.e, there are at most 2^{21} distinct possible values of the bitwise XOR of a path.

Now, recall the **pigeonhole principle**: if the number of pigeons (paths) exceeds the number of holes (XOR values), then two pigeons must share a hole (two paths must share a XOR value).

In particular, this means that if we take $2^{21} + 1$ different paths and compute their bitwise XORs, we will definitely find two paths with the same XOR.

This gives us a very simple solution: brute force!
Simply look at paths in the tree one at a time, each time computing its bitwise XOR.
You will either find two paths with equal bitwise XOR, or run out of paths.

The latter can only happen when the number of paths in the tree doesn't exceed 2^{21}, so each testcase is solved in at most 2^{21} operations, which is good enough.

Computing the bitwise XOR of a given path is not hard, and can be done in a couple of ways.

- The simplest way is to do it on-the-fly. Fix one end of the path, then perform a DFS/BFS on the tree to compute bitwise XORs to all other vertices; immediately stopping once you reach a duplicate.
- Alternately, there's a more general technique. Fix a root of the tree, say vertex 1, and compute the bitwise XOR of the $1 \to u$ path for every u; say this value is X_u.
  Then the bitwise XOR of the path between u and v is simply $X_u \oplus X_v, since the XORs of edges not on the path cancel out.

TIME COMPLEXITY

$\mathrm{O}(\min(N^2, \mathrm{MAX}))$ per test case, where $\mathrm{MAX} = 2^{21}$ for this problem.

PROBLEM:

You're given an integer N.
For an integer $1 \le x \le N$, consider the following process:

- Pick a random integer y between 1 and N
- Replace x by $\gcd(x, y)$.

Let $\mathrm{E}[x]$ be the expected number of moves this process takes till x reaches 1.
For each $1 \le x \le N$, compute $\mathrm{E}[x]$.

EXPLANATION:

Writing out the expected value for a single x, we see that:

$$\mathrm{E}[x] = \sum_{y=1}^{N} \frac{1}{N} \cdot (1 + \mathrm{E}[\gcd(x, y)]) = 1 + \frac{1}{N} \sum_{y=1}^{N} \mathrm{E}[\gcd(x, y)]$$

This lends itself to a rather natural (if slow) recursive approach, because $\gcd(x, y) \le x$.
The only issue is those y for which $\gcd(x, y) = x$, since they lead to 'loops'.

However, such y lead to an $\mathrm{E}[x]$ term anyway, so they can be taken care of with a bit of algebra.
Suppose there are k such y. Then,

$$\mathrm{E}[x] = 1 + \frac{k}{N}\mathrm{E}[x] + \frac{1}{N} \sum_{\gcd(x,y) \neq x} \mathrm{E}[\gcd(x, y)]$$
$$\mathrm{E}[x] \cdot (1 - \frac{k}{N}) = 1 + \frac{1}{N} \sum_{\gcd(x,y) \neq x} \mathrm{E}[\gcd(x, y)]$$

The right-hand side here now depends only on values strictly less than x, so a dynamic programing approach can compute it.
Once the RHS is known, $\mathrm{E}[x]$ is easily found.
However, it's still $\mathrm{O}(N^2)$, so we need to speed it up.

Let's look at the expression we want to compute.
The expression we want to compute is $\sum_{\gcd(x,y) \neq x} \mathrm{E}[\gcd(x, y)]$.

Let's fix $g = \gcd(x, y)$ and try to find the number of $1 \le y \le N$ such that $\gcd(x, y) = g$.
If we know this, we can add this count multiplied by $\mathrm{E}[\gcd(x, y)]$ to the overall sum.
g has to be a divisor of x, which greatly limits the number of g we need to consider.

Writing $x = g \cdot s$ and $y = g \cdot t$, we have $\gcd(g \cdot s, g \cdot t) = g$, which means $\gcd(s, t) = 1$.
Further, $g \cdot t \le N$, meaning $t \le \frac{N}{g}$

So, we really just want to compute the number of integers in the range $[1, \frac{N}{g}]$ that are coprime to $s = \frac{x}{g}$.

That can be done using inclusion-exclusion on the prime factors of s.

Details

Let $M = \frac{N}{g}$ be our upper bound.
Let the distinct prime factors of s be $p_1, p_2, \ldots, p_k$.

Then, we have the following:

- Initially, pretend everything in $[1, M]$ is coprime to s.
- Anything not coprime to s must have some p_i as a factor.
  So, subtract all multiples of $p_1, p_2, \ldots, p_k$ from the count.
  These are $\lfloor \frac{M}{p_1} \rfloor, \lfloor \frac{M}{p_2} \rfloor, \ldots, \lfloor \frac{M}{p_k} \rfloor$ respectively.
- However, now multiples of, for example, $p_1 p_2$ were counted twice.
  So, for each pairwise product of primes, add back in the count of its multiples.
- Then, you'll need to subtract the count of multiples of all triplet products, and so on.

In general, we go through each of the 2^k possibilities of products of prime factors, and either add or subtract the count of its multiples depending on how many primes are involved in the product.

A slightly more detailed explanation can be found **here**.

Notice that if s has k distinct prime factors, this approach takes 2^k or $2^k \cdot k$ steps, depending on implementation.
Since $N \leq 5 \cdot 10^5$, any integer $\leq N$ has at most 6 distinct prime factors, so $k \leq 6$ here, which is pretty small.

We do the 2^k operation for each pair of (number, divisor) below N.
It's well-known that there are $\mathrm{O}(N \log N)$ such pairs, so a simple upper bound on complexity is $\mathrm{O}(2^6 \cdot N \log N)$.

However, in practice most integers have far fewer than 6 prime factors, so the constant factor is a lot better and your code should run fast.
In particular, even with a $2^k \cdot k$ algorithm that goes through bitmasks, the number of operations is about 10^8 which is not an issue for C++.

A 2^k algorithm for each pair uses less than $4 \cdot 10^7$ operations.

TIME COMPLEXITY

$\mathrm{O}(2^6 \cdot N \log N)$ per testcase, but with fairly low constant factor.

PREREQUISITES:

Harmonic series/sieve-like methods, prefix sums/binary search

PROBLEM:

You have an array A, where $1 \le A_i \le M$ for each i denotes the budget of the i-th customer.

There are infinitely many candies; whose price you can choose to be between 1 and M (every candy has the same price).
The i-th customer will buy as many candies as their budget allows.

You also have an array C of length M, where C_i denotes the bonus you receive for each candy of price i bought.
Find the maximum possible bonus if the price is chosen appropriately.

EXPLANATION:

Suppose we fix the price of the candy, P.

Of course, we can always look at each element of A and figure out how much each person will buy, but that's too slow.
Let's look at a different perspective.

How many people will buy exactly k candies when the price is P?
It's not hard to see that this is exactly the number of people whose A_i lies in the range $[kP, (k+1) \cdot P - 1]$.

If we were able to quickly count the number of people whose A_i lies in this range (say, in $\mathrm{O}(1)$), then we could iterate over every possible value of k and add $k \cdot \mathrm{count}$ to the number of candies we sell.

Computing this count quickly is fairly simple.

How?

If we sort the A_i, finding the number of elements in a given range is a simple exercise in binary searching, and can be done in $\mathrm{O}(\log N)$.

In fact, we can utilize the constraints to do even better.
Notice that $A_i \le M$.

So, let freq be an array of length M, where $\mathrm{freq}[r]$ denote the number of people with $A_i = r$.
Then, what we want is $\mathrm{freq}[kP] + \mathrm{freq}[kP+1] + \ldots + \mathrm{freq}[kP+k-1]$.
This is a range sum on freq, which can be computed in $\mathrm{O}(1)$ using prefix sums.

Now, notice that for a fixed P, we don't need to check too many values of k.
In particular, we can stop as soon as $M < kP$, because $A_i \le M$ anyway.
This means we need to check for each k from 1 to $\lfloor \frac{M}{P} \rfloor$.
Each check is done in $\mathrm{O}(1)$.

Doing this for every P from 1 to M brings our overall time complexity to

$$\mathrm{O}\left(\sum_{P=1}^{M} \lfloor \frac{M}{P} \rfloor\right)$$

which is, rather famously , equal to $\mathrm{O}(M \log M)$.

TIME COMPLEXITY:

$\mathrm{O}(M \log M)$ per testcase.

PROBLEM:

You're given an undirected graph with N vertices and M edges.
Vertex i has the value A_i written on it.

Find the minimum cost of choosing a connected subset of K vertices from this graph, where the cost of choosing vertices $x_1, x_2, \ldots, x_K$ is

$$(A_{x_1} \mid A_{x_2} \mid \ldots \mid A_{x_K}) - (A_{x_1} \,\&\, A_{x_2} \,\&\, \ldots \,\&\, A_{x_K})$$

Print -1 if no such subset exists.

EXPLANATION:

Let's rewrite the cost a bit.
Notice that $(A_{x_1} \,\&\, A_{x_2} \,\&\, \ldots \,\&\, A_{x_K})$ is always a *submask* of $(A_{x_1} \mid A_{x_2} \mid \ldots \mid A_{x_K})$.
So, the difference between them is also just their XOR.
Thus, the cost can be written as

$$(A_{x_1} \mid A_{x_2} \mid \ldots \mid A_{x_K}) \oplus (A_{x_1} \,\&\, A_{x_2} \,\&\, \ldots \,\&\, A_{x_K})$$

Everything's a bitwise operation now, so we can look at bits one-by-one.
In particular, a bit b is present in the cost if and only if:

- At least one of the A_{x_i} has bit b set
- At least one of the A_{x_i} has bit b unset

Our aim is to minimize the cost. Since we're dealing with bits one at a time, this can be attempted greedily — that is, if we can avoid taking bit b even at the cost of taking *all* of bits $0, 1, \ldots, b-1$; then it's better to do so.

With this in mind, let's attempt a greedy solution.

Initially, we have all the vertices available to us. Let this set be V.
Let's look at $b = 29$, the highest possible bit.
If we are to *avoid* having b in the answer, then:

- Either *all* the vertices we choose must not have bit b set; or
- *All* the vertices we choose must have bit b set.

Notice that this splits us into independent problems.
That is, if L is the subset of V whose elements don't have b set; and R is the subset of B whose elements do have b set, we're essentially trying to recursively solve for both L and R, independently.

If any one of them have a solution, we can definitely avoid taking b into the answer.
The issue is when neither of them have solutions: then, we *must* have 2^b present in the answer.
When this is the case however, we can simply try to partition based on the next bit; after all, that's what the greedy approach would do.

This gives us a recursive formulation:
Let $f(V, b)$ be the answer for the vertex set V, if we're considering bit b.
$f(V, b)$ returns ∞ (in implementation, some sufficiently large number) if we can't choose a connected subset of size K from V.

Let L be the subset of V that doesn't have b set, and R be the other elements.
Then,

- First, set $f(V, b) = \min(f(L, b-1), f(R, b-1))$ recursively.
- Now, if $f(V, b) \neq \infty$, this is the correct value of $f(V, b)$ (and in particular doesn't include 2^b) so we can just return it.
- Otherwise, return $2^b + f(V, b-1)$.

We need to take care of a couple of details: mainly the base cases for the recursion.

- First, let's decide if $f(V, b) = \infty$ or not.
  This is (theoretically) quite simple: consider only vertices from V, and edges between these vertices. If the resulting graph has a connected component of size $\geq K$ then the answer is not ∞; otherwise there is no way to choose K connected vertices so we have $f(V, b) = \infty$.
  If $f(V, b) = \infty$ you can return it immediately, no further processing required.
  Returning ∞ early like this is important!
- Next, the base case: if $b = -1$, we've processed all the bits, so we can return 0 immediately.

With proper implementation care, this is indeed enough to get AC (its time complexity seems a bit dubious - we're recursing with sets?).

As for implementation details; the main thing to be careful about is the connected component size check — it can be done using BFS/DFS and directly iterating across the elements of V and their neighbors, just be careful not to create auxiliary arrays of size N each time (to denote marked/visited vertices and such); instead you can keep global arrays and only use the cells you need.

Now, let's look at how fast this actually is.

Let's look at the recursion tree, level by level.
At the top level, we have all the vertices and $b = 29$.
Next, one of two things happens:

- We create sets L and R and check if $f(L, 28)$ and $f(R, 28)$ are ∞ or not.
- If one (or both) are not ∞, then we recurse into them. However, across the entire second level, each vertex appears only at most once since L and R are disjoint.
- If both are ∞, we remain with the same set when recursing: again, each vertex appears only once.

It's not hard to see that this is maintained: as we go down the tree, any particular level will still contain every vertex only at most once.
There are only 30 levels, so this isn't really an issue.

For the same reason, all the DFS/BFS-es we're doing to find connected components also aren't an issue.
Across each level, each edge will only be considered at most twice - once for each of its endpoints. Whether the edge is ignored or not doesn't matter.
So, the connected component checks across an entire level still take $\mathrm{O}(N + M)$ time, leading to $\mathrm{O}(30 \cdot (N + M))$ time in total, which is good enough.

It's possible to implement this algorithm iteratively, which makes the time more obvious.

Let's keep S: a list of sets of vertices; and the answers corresponding to each set.
Initially, S contains only the single set $\{1, 2, \ldots, N\}$. The answer corresponding to it is $2^{30} - 1$, i.e, it contains all 30 bits.

Then, for each b from 29 down to 0:

- Iterate over all the sets X in S. For the set X whose answer is val_X,
  - Split X into L and R, based on bit b.
  - If L contains a component of size $\geq K$, add L into S; with its corresponding answer being $\mathrm{val}_X \oplus 2^b$.
  - If R contains a component of size $\geq K$, add R into S; with its corresponding answer being $\mathrm{val}_X \oplus 2^b$.
  - If either L or R were added to S, then X can be deleted from S.

- The final answer is the maximum answer among all sets remaining in S; and `-1` if no sets remain.

This way, it should be obvious that at each bit we do $\mathrm{O}(N+M)$ work, for $\mathrm{O}(30 \cdot (N+M))$ in total.

TIME COMPLEXITY

$\mathrm{O}(B \cdot (N+M))$ per test case; where $B = 30$ for this problem.

PREREQUISITES:

Familiarity with probability, (optional) **solving linear recurrences**

PROBLEM:

Two players play a competition with N questions, each with M options.
The competition proceeds as follows:

- On their turn, the contestant will choose an answer to the current question uniformly randomly from among the remaining options.
- If their answer is correct, they move to the next question and keep their turn.
- If their answer is wrong, the turn passes to the opponent.

The winner is whoever answers question N.
Find the probability that player 1 wins.

EXPLANATION:

Let's first look at a slow but correct solution, then see how to speed it up.

Let P_i denote the probability that player 1 is the first to answer question i correctly.
Our aim is to compute P_N.

P_i can be defined in terms of a recurrence.
Either player 1 answered question $i-1$ correctly, or player 2 did.
Let's separate these cases and calculate them individually.

Suppose player 1 answered question $i-1$ (which has a probability of P_{i-1} of happening), and hence goes first on question i.
Then, the probability that he answers question i correctly is

$$\frac{1}{M} + (\frac{M-1}{M} \cdot \frac{M-2}{M-1} \cdot \frac{1}{M-2}) + (\frac{M-1}{M} \cdot \frac{M-2}{M-1} \cdot \frac{M-3}{M-2} \cdot \frac{M-4}{M-3} \cdot \frac{1}{M-4}) + \dots$$

because:

- There's a $\frac{1}{M}$ chance of getting it right on the first try.
- If the first try fails ($\frac{M-1}{M}$ chance) , the second player must also get it wrong ($\frac{M-2}{M-1}$ chance, there's one less option now) and then there's a $\frac{1}{M-2}$ chance of player 1 getting it right on his second guess.
- If that fails, once again the second player should get his second guess wrong, and so on.

However, note that each term above simply cancels out into $\frac{1}{M}$.
So, the required probability is simply the sum of $\frac{1}{M}$ several times.
In particular, notice that we obtain one $\frac{1}{M}$ term for every odd number $\leq M$, of which there are exactly $\lceil \frac{M}{2} \rceil$.

Similarly, for the case when player 2 answered question $i-1$ first, we see that the required probability is

$$(\frac{M-1}{M} \cdot \frac{1}{M-1}) + (\frac{M-1}{M} \cdot \frac{M-2}{M-1} \cdot \frac{M-3}{M-2} \cdot \frac{1}{M-3}) + \dots$$

which is once again the sum of $\frac{1}{M}$,but this time $\lfloor \frac{M}{2} \rfloor$ of them — one for each even number $\leq M$.

Putting the cases together, we obtain

$$P_i = P_{i-1} \times (\lceil \frac{M}{2} \rceil \cdot \frac{1}{M}) + (1 - P_{i-1}) \times (\lfloor \frac{M}{2} \rfloor \cdot \frac{1}{M})$$

This holds for all $i > 1$, and even holds for $i = 1$ if we define $P_0 = 1$ (since player 1 always starts on question 1, equivalently we can instead pretend player 1 always answers question 0).

The above formula is a linear recurrence!
That is, it's of the form $P_i = x \cdot P_{i-1} + y$ for some constants x and y.

Finding the N-th term of such a recurrence can be done in $\mathrm{O}(\log N)$ using matrix exponentiation, as seen in this **blogpost** for example.

This is already enough to solve the problem, but read on if you'd like to see a solution that doesn't rely on matrices!

Notice that our formula deals with floor and ceiling division of M by 2.
Let's look at what happens to even and odd M separately.

Even M

Suppose $M = 2k$. Then, $\lfloor \frac{M}{2} \rfloor = \lceil \frac{M}{2} \rceil = k$.
So, our formula becomes

$$P_i = P_{i-1} \cdot k \cdot \frac{1}{M} + (1 - P_{i-1}) \cdot k \cdot \frac{1}{M} = \frac{k}{M}$$

But $M = 2k$, so $\frac{k}{M} = \frac{1}{2}$.

That is, if M is even the answer is simply $\frac{1}{2}$, irrespective of M.

Odd M

Suppose $M = 2k + 1$, so $\lfloor \frac{M}{2} \rfloor = k$ and $\lceil \frac{M}{2} \rceil = k + 1$.
Plugging into the formula,

$$\begin{aligned} P_i &= P_{i-1} \cdot (k+1) \cdot \frac{1}{M} + (1 - P_{i-1}) \cdot k \cdot \frac{1}{M} \\ &= \frac{P_{i-1}}{M} + \frac{k}{M} \end{aligned}$$

Repeatedly applying this formula, along with the base case $P_0 = 1$, gives us

$$P_i = \frac{1}{M^i} + k \cdot (\frac{1}{M} + \frac{1}{M^2} + \ldots + \frac{1}{M^i})$$

The quantity inside the brackets is the sum of a **geometric progression**, and can be quickly computed using a formula for the same.
This allows for easy computation of P_N.

We've thus found a closed-form formula for P_N, and we're done!

TIME COMPLEXITY

$\mathrm{O}(\log MOD)$ per test case.

PREREQUISITES:

Sorting

PROBLEM:

A restaurant has N dishes; the i-th of which has category A_i and takes B_i time to be cooked.
Find the minimum possible time needed to get at least K distinct categories of food.

EXPLANATION:

If there are less than K distinct categories the answer is -1.
This can be checked in several different ways, for example:

- Put all the A_i values into a set and check its size; or
- Sort the A_i values and count the number of indices such that $A_i \leq A_{i+1}$.

Now let's deal with the case when there are $\geq K$ distinct values.

Choosing two different dishes with the same category is clearly not optimal.
This means we choose at most one dish from each category.

For the least total time, from each category we obviously keep only the one that takes the least time to cook.

This can be computed using an array C of length 10^5, initially $C_i = \infty$ for each $1 \leq i \leq 10^5$.
Then, for each $1 \leq i \leq N$, set $C_{A_i} = \min(C_{A_i}, B_i)$.

Now, C_x contains the minimum time to cook a dish of category x.
The answer is then the sum of the K smallest values of C, which can be found by sorting C.

Instead of an array of size 10^5, you can also use a map (`std::map` in C++, `HashMap/TreeMap` in Java, `dict` in Python) to represent C.

TIME COMPLEXITY:

$\mathrm{O}(N \log N)$ per testcase.

PROBLEM:

You're given a tree with N vertices; the i-th has value v_i.
Support the following updates and queries on it:

1. Given u and p, multiply the values of all values in the subtree of u, by v_i.
2. Given u and p, fully divide the values of all values in the subtree of u, by v.
   Here, fully divide means "while x is divisible by p, divide it by p".
3. Given u, count the number of primes that divide all values in the subtree of u.

it is known that $p, v_i \leq 700$.

EXPLANATION:

The third query suggests that looking at the prime factorizations of the values is helpful.
Looking at the how the updates affect values in terms of their factorizations:

- The first update means "all values in the subtree of u now have p as a prime factor".
- The second update means "all values in the subtree of u no longer have p as a prime factor".

This means the prime factorizations of each element are rather boolean in nature: for each element, we only need to know which primes exist; we don't care about their powers.

Since $p, v_i \leq 700$, there are only 125 primes we care about in total.
So, let's represent the prime factorization of each element by a 125-bit bitmask.
Let M_i denote the mask for vertex i.
This bitmask can be represented using two 64-bit integers in most languages (or, in C++, a `bitset<125>`).

Translating our updates and queries to this bitmask:

- Update 1 means "set a certain bit to 1 for all masks in the subtree of u".
  This can be represented by a bitwise OR operation.
- Update 2 means "set a certain bit to 0 for all masks in the subtree of u".
  This can be represented by a bitwise AND operation.
- Update 3 means "find the number of bits that are set in all masks in the subtree of u".
  This can be thought of as "find the bitwise AND of all masks in the subtree of u, then count the number of set bits".

We now need to perform these efficiently.

First off, as always we can transform subtree operations into subarray operations by flattening the tree via an Euler tour.
Now, we need to support the following. We have an array M of length N. Then,

- Given (L, R, x), set $M_i \to M_i \mid x$ for all $L \leq x \leq R$.
- Given (L, R, x), set $M_i \to M_i \,\&\, x$ for all $L \leq x \leq R$.
- Given (L, R), compute the bitcount of $(M_L \,\&\, M_{L+1} \,\&\, \ldots \,\&\, M_R)$.

Since we're performing lazy propagation, we need to be able to combine updates; and this is somewhat non-trivial because there are two completely different types of updates.

However, it is indeed possible to do so.
For each bit, there are three possible updates to it:

- Keep it the same
- Set it to 1
- Set it to 0

So, each update is a *ternary* state.
Combining these ternary states, however, can be done using just binary operations as follows:

- Each update type consists of two bitmasks x and y.
  By default, x has all its bits set and $y = 0$.
- Then, to push an update (x_1, y_1) from a parent node in the segment tree to its child whose update is currently (x_2, y_2), do the following:
  - Set $x_2 \to (x_2 \ \& \ x_1) \mid y_1$
  - Set $y_2 \to (y_2 \ \& \ x_1) \mid y_1$

This process of performing the AND update before the OR one is always correct, and can be verified manually against all 3×3 possibilities of combining updates for a single bit.

Since we're using purely bitwise operations, this is now just standard lazy propagation on a segment tree whose time complexity is $\mathrm{O}(Q \log N)$, which is fast enough.

TIME COMPLEXITY

$\mathrm{O}(N + Q \log N)$ per testcase.

PREREQUISITES:

Familiarity with bitwise operations

PROBLEM:

You're given two functions f and g, whose definitions can be found in the statement.
You're also given integers L and R.

Find the maximum value of $f(x) + g(x)$ across all $L \leq x \leq R$.

EXPLANATION:

Our first order of business should be figuring out what the functions f and g actually are, since they're given to us in a recursive form.

This can be done by analyzing them a bit (or computing a few early values and putting the sequence into **OEIS** 😛)

Computing f(x)

Looking at the definition of f should make you think of binary, since computing it requires us to divide by 2 each time.

You should quickly notice that, in terms of binary:

- If the last digit is 0 (meaning the number is even), we add 1 to the answer; otherwise we add 0 to the answer.
- Then, we delete the last digit (which corresponds to dividing by 2)

In other words, $f(x)$ is simply the number of zeros in the binary representation of x.

Computing g(x)

Looking at the definition of g, we see the following:

- Suppose $x = 2k$. Then, $g(x) = 2 \cdot g(k) + 1$.
- Suppose $x = 2k + 1$. Then, $g(x) = 2 \cdot g(k)$.

Notice that these are suspiciously similar: obtaining x from k is exactly the *opposite* of obtaining $g(x)$ from $g(k)$; in terms of adding 1 after multiplying by 2.

Now let's contextualize this in terms of binary.
Any integer x can be obtained by starting with 1, then continually multiplying by 2 and adding either zero or one, depending on whether we need to add a new bit or not.

We can thus build $g(x)$ as we build x; except that we add a bit to $g(x)$ if and only if we *don't* add a bit to x.

In other words, $g(x)$ is obtained by *inverting all the bits of* x, except for the highest bit.

For example, if $x = 17 = 10001_2$, then $g(x) = 11110_2 = 30$.

Now that we know f and g, let's observe a couple of their properties.

- $f(x)$ is pretty small: since $x \leq R \leq 10^9$, we have $f(x) \leq 30$.
- $f(x)$ is largest when x is a power of 2.
- If $x + 1$ is not a power of 2, then $g(x) > g(x + 1)$.
  - In fact, if $x + 1$ is not a power of 2, then $g(x) = g(x + 1) + 1$.

- If $x+1$ is a power of 2, then $g(x) < g(x+1)$.
  - In this case, $g(x+1)$ is larger than $g(y)$ for every $y \le x$.

The above discussion tells us that to maximize $f(x)+g(x)$, ideally x should be a power of 2.
So, if the range $[L, R]$ contains a power of 2, simply choose x to be the largest power of 2 in this range and compute $f(x)+g(x)$ for it — that's the answer.

Now, what about when $[L, R]$ doesn't contain a power of 2?
Notice our third point above: g is a strictly decreasing function on this range, i.e, $g(L) > g(L+1) > \ldots > g(R)$.
Also recall that f can only take values upto 30.

So, for any $x > L+31$, $f(x)+g(x)$ can never be larger than $f(L)+g(L)$, since f cannot make up for the difference between the g-values.

So, it's enough to check for $x = L, L+1, \ldots, L+31$.
Compute $f(x)+g(x)$ for each one, take the maximum of them all.

Computing $f(x)$ and/or $g(x)$ for a given x can be done in $\mathrm{O}(\log x)$ using the given recursive definitions.

TIME COMPLEXITY:

$\mathrm{O}(\log^2 R)$ per testcase.

PROBLEM:

Given a complete graph on N vertices and two of its vertices u and v, find the maximum bitwise XOR of a path from u to v, and any one shortest path that attains this maximum.

EXPLANATION:

The bitwise XOR operation is associative, so the order in which our path visits vertices doesn't really matter — we only need to ensure that no vertex is visited more than once; and that u and v are visited.
This means we only care about *subsets* of vertices — their order can be fixed afterwards.

In fact, this observation is enough to solve **subtask** 1.
$N \leq 20$, so simply check every subset of vertices that contains both u and v, and take the best one (maximum bitwise XOR, then minimum size).

Let N be a $(k+1)$-bit integer, i.e, 2^k is the highest power of 2 in its binary representation.
Clearly, the maximum possible XOR attainable is $2^{k+1} - 1$; so we'd like to figure out:

1. Whether it's possible to reach this
2. The shortest path to reach it

Time for a bit of case analysis!
Let $M = 2^{k+1} - 1$. Then,

- If $u \oplus v = M$, we're done with a length-2 path
- If not, let's check for a length-3 path.
  For this, we'd need a w such that $u \oplus v \oplus w = M$, or $w = M \oplus u \oplus v$.
  If this w is valid (i.e, is $\leq N$ and equals neither u nor v), we're done.
  Otherwise, there are a couple of points of failure; so let's see what happens.

If $w \leq N$, then we run into issues only if $w = u$ or $w = v$.
Without loss of generality, let $w = u$.
Then, along with $u \oplus v \oplus w = M$, we'll have $v = M$ (which also means that $N = M$).

So, all we need to do is find two vertices x and y (distinct from u and v) such that $x \oplus y = w$; then we can take the path $u \to x \to y \to v$.
This is fairly easy because $N = M$ and we have only two 'bad' vertices; for example either the pair $(1, w \oplus 1)$ or the pair $(2, w \oplus 2)$ will work.

So far, we've always been able to attain a maximum of M; and have used at most 4 vertices (including u and v) to do so.
Notice that this is enough to solve **subtask** 2; since $N = 2^{k+1} - 1 = M$ is guaranteed here.

Now, let's look at the remaining case when we can't reach M using ≤ 3 vertices, i.e, $w > N$.
We want to select a minimum-sized subset of integers from 1 to N (excluding u and v), such that their XOR equals w.
Note that $w > N$, which means w has 2^{k-1} set in its binary representation.
In particular, this means that the subset we choose should certainly include an integer $\geq 2^k$.

Suppose $N = 2^k + r$, where $0 \leq r < 2^k$.
Then, if r is large enough, finding a pair of valid vertices whose XOR is w is not hard.
In particular, if $r \geq 2$ then it can be shown that at least one of the pairs $(2^k, 2^k \oplus w), (2^k + 1, (2^k + 1) \oplus w), (2^k + 2, (2^k + 2) \oplus w)$ will work, so path length $= 4$ in these cases.

This leaves us with $r = 0$ and $r = 1$ to deal with.

r = 0

When $r = 0$, we have $N = 2^k$.
Note that since $u, v \le N$ and $w > N$, we *can't* have $u = 2^k$ or $v = 2^k$ once we're in this case — that means $u, v < N$.

Let's check if a size-4 subset is possible.
For this, recall from earlier that we *must* choose something with 2^k set; and our only option is to choose 2^k itself.

So, check if $(2^k, w \oplus 2^k)$ works.
If it does, great, we're done in 4 moves.
Otherwise, since we know $u, v \neq 2^k$, we only failed because either u or v equals $2^k \oplus w$. WIthout loss of generality, let it be u.

Along with $u \oplus v \oplus w = M$, we have $v = 2^k - 1$.
Now that u and v are fixed, a subset of size 5 works: (at least) one of $(2^k, 1, u \oplus 1)$ or $(2^k, 2, u \oplus 2)$ will work.

r = 1

Just as we did in the $r = 0$ case, let's check if a subset of size 4 works.
There are only two possibilities for the pair here: $(2^k, 2^k \oplus w)$ and $(2^k + 1, (2^k + 1) \oplus w)$.
Check both, and if one of them works we're done.

If they both fail, we have two possibilities: $u, v < N$, or $(u, v) = (2^k, 2^k + 1)$.
The $u, v < N$ case is similar to how it was in $r = 0$, so the answer equals 5 there.

The $(u, v) = (2^k, 2^k + 1)$ case is a bit more interesting: it's the only case when we can't achieve M as the maximum!
This is because $u \oplus v = 1$ and there are no more vertices with 2^k set that we can choose from.
So, in this case alone, the maximum XOR is in fact $2^k - 1$, and can be achieved with a subset of size 3.

This takes care of all the cases.

Some of the discussion above doesn't work when N is very small, because having too few vertices to choose from constraints freedom a bit.
The safest way to do this is to just bruteforce for small enough N, and apply the casework for larger ones.
The casework above definitely works for $N \ge 8$, so bruteforcing when $N \le 7$ is good enough.

Implementing this might seem painful with all the cases we have, but there's a fairly simple way to implement the solution:

- First, do a brute for small N.
- Then, check if the maximum can be M (which is almost always the case).
- For this,
  - Checking if 2 or 3 vertices are enough is trivial
  - Checking if 4 vertices are enough (i.e, $\{u, v, x, y\}$) can be done in $\mathrm{O}(N)$: iterate over all choices of x, which also fixes y and then their validity can be checked.
  - Next, check if 5 vertices are enough.
    This can also be done in $\mathrm{O}(N)$ time by noting that whenever the answer is 5, there's always a solution that picks either vertex 1 or 2.
    So, fix one of these and then run the $\mathrm{O}(N)$ solution for four vertices
- If all the above fail, we're in the special case where the answer is $< M$, and that can be solved using three vertices as mentioned above.

This way, you only need to write a handful of loops and if statements.

TIME COMPLEXITY

$O(1)$ per test case.

PROBLEM:

Given N and K, determine whether it's possible to partition the integers $\{1, 2, \ldots, N\}$ into exactly K groups such that:

- Each group contains ≥ 2 integers.
- The sum of each group is odd.

EXPLANATION:

First off, note that we want to form K groups with ≥ 2 integers each.
This means we need at least $2K$ integers in the first place; so if $N < 2K$ the answer is immediately `No`.
Now let's analyze when $N \geq 2K$.

We have $x = \lceil \frac{N}{2} \rceil$ odd integers and $y = \lfloor \frac{N}{2} \rfloor$ even integers to work with.
Note that we'll have $x = y$ if N is even, and $x = y + 1$ if N is odd.
Also, $N \geq 2K$ means $x, y \geq K$.

The sum of each group must be odd, which means each group should contain an odd number of odd integers.
The even integers don't affect the parity of the sum, so they can be distributed however we like.

Since $x, y \geq K$, let's first put one odd and one even integer into each group.
Now, each group has odd sum and ≥ 2 elements; we just need to figure out whether the other elements can be distributed properly while maintaining this property.

We're left with $x - K$ odd integers, which need to be distributed among the K groups.
To keep the sum of each group odd, note that even group must receive an even number of these $x - K$ integers.
In particular, this means $x - K$ itself must be even; and it's not hard to see that this condition is also sufficient (if $x - K$ is even, give all $x - K$ odd integers to the first group and we're done).

So, when $N \geq 2K$, the answer is `Yes` if $x - K$ is even and `No` otherwise.

TIME COMPLEXITY

$\mathrm{O}(1)$ per test case.

PREREQUISITES:

Sieve of eratosthenes

PROBLEM:

Given N, partition the integers $\{1, 2, \ldots, 2N\}$ into N pairs such that number of pairs (x, y) with $\gcd(x, y) = 1$ is minimum.

EXPLANATION:

First, let's look at which pairs unavoidably must have $\gcd = 1$.

- $\gcd(1, x) = 1$ for any integer x, so the pair containing 1 is sad.
- If p is a *prime number* such that $N + 1 \le p \le 2N$, then $\gcd(p, x) = 1$ for any other integer between 1 and $2N$.

Every other number has either a factor or a multiple within the range $[2, 2N]$, so there's potential to form a non-coprime pair using it.

This immediately gives us a lower bound on the number of sad pairs: if there are k primes between $N + 1$ and $2N$, then there are at least $\lceil \frac{k+1}{2} \rceil$ sad pairs; at best these primes and 1 can be paired with each other.

In fact, this lower bound is also tight!
We'll prove this by construction.

Notice that if $\gcd(x, y) > 1$, there must exist a prime p that divides both x and y.
Of course, with respect to the problem we're dealing with, p must also lie between 2 and N (since we've already seen that primes $> N$ are coprime with everything else).

So, let $p_1 < p_2 < \ldots < p_m$ be the distinct primes between 2 and N.
Then, we can do the following:

- Consider these primes in descending order, i.e, p_m down to p_1.
- For a fixed prime p_i,
  - Let L_i denote the list of all *multiples* of p_i that haven't yet been paired.
    Note that L_i surely contains at least p_i and $2p_i$; though it might contain more elements as well.
  - If L_i has even size, pair off all its elements within themselves; which is fine since any pair created this way will have a gcd that's a multiple of p_i.
  - If L_i has odd size, we need to discard one element and pair the rest.
    Here, discard $2p_i$ and pair everything else up.

At the end of this process, most elements have been paired up with some others to form $\gcd > 1$ pairs.
The only unpaired elements are the ones we discarded when L_i had odd size — however, note that we chose $2p_i$ each such time, i.e, all these elements are even!
So, we can pair them among themselves without issue.

Finally, we're left with at most one unpaired element, along with 1 and all the primes from $N + 1$ to $2N$.
Pair them up arbitrarily since the gcd of any pair will be 1.

It's not hard to see that this process creates exactly $\lceil \frac{k+1}{2} \rceil$ pairs with $\gcd = 1$, since only the very last step creates such pairs.
Since that was our lower bound for sad pairs, and we've attained it, optimality is proved.

As for implementation, there are only two nontrivial parts here:

- Finding all primes upto $2N$
- Finding the lists L_i, i.e, unpaired multiples of prime p_i.

The first part can be done in $\mathrm{O}(N \log N)$ (or better, depending on implementation) using the sieve of Eratosthenes.

The second part can be done in a brute-force manner, since iterating across all multiples of p_i for each prime upto N has a time complexity of $\mathrm{O}(N \log \log N)$ (and is very similar to the aforementioned sieve anyway).

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PROBLEM:

Given a simple polygon with N vertices and perimeter at most 10^7, count the number of lattice points (x, y) on its perimeter such that $(x + y)$ is prime.

The coordinates of the points can be as large as 10^{14}.

EXPLANATION:

The perimeter of the polygon is at most 10^7, so there are certainly at most 10^7 lattice points lying on it.
Let's first try to find all such lattice points.

The polygon is simple and we're given its vertices in order, so the perimeter can be broken up into a bunch of line segments.
It suffices to find lattice points lying on each such segment.

So, consider the segment from (x_1, y_1) to (x_2, y_2).
Then, if $d = \gcd(x_2 - x_1, y_2 - y_1)$, there are exactly $g + 1$ lattice points lying on this segment (including the endpoints); in fact their coordinates will be of the form

$$(x_1 + k \cdot \frac{x_2 - x_1}{d}, y_1 + k \cdot \frac{y_2 - y_1}{d})$$

for $0 \le k \le d$.
A quick explanation of why this is the case can be found **here**.

This allows us to easily find all the lattice points lying on the perimeter in $\mathrm{O}(N + P)$ (where P is the perimeter).
Now, we need to test each one for primality.

A brute-force primality check for an integer α would take $\mathrm{O}(\sqrt{\alpha})$ time, which is obviously too slow here since we need to run it several times.

If the maximum value of α were small enough (say, $\le 10^7$), we could use a **sieve of Eratosthenes** to precompute all primes upto the limit and then answer queries in $\mathrm{O}(1)$.
However, here α can be as large as $2 \cdot 10^{14}$, so this has no chance of working.

The important observation here comes once again from the fact that the perimeter is small: although the values themselves might be large, the *range* of values that we need to check primality for is quite small!
In particular, let m_x and M_x be the smallest and largest x-coordinates of some lattice points, respectively.
SImilarly define m_y and M_y.
Then, the smallest integer we care about is $m_x + m_y$, and the largest is $M_x + M_y$.

However, since the perimeter is $\le 10^7$, we have $M_x - m_x \le 10^7$ and $M_y - m_y \le 10^7$, and so $(M_x + M_y) - (m_x + m_y) \le 2 \cdot 10^7$.

This range being small allows us to use a modification of the sieve of Eratosthenes known as the **segmented sieve**, to compute primality of everything in this range.

The idea to precompute primes in range $[L, R]$ is as follows:

- First, use a normal sieve to compute all the primes upto $\sqrt{R}$.
- Then, for each prime p, mark all of its multiples in range $[L, R]$ as *not prime*.
- Finally, everything in range that wasn't marked is a prime.

This works because every non-prime in this range will have a prime factor that's $\le \sqrt{R}$.
The time complexity of this is $\mathrm{O}((R - L + 1 + \sqrt{R}) \log \log R)$.

Notice that this immediately solves the problem: the segmented sieve precomputes all primes we care about, then primality of each lattice point can be checked in $\mathrm{O}(1)$ time.

TIME COMPLEXITY

$\mathrm{O}(N + P + (R - L + 1 + \sqrt{R}) \log \log R)$ per test case, where P is the perimeter of the polygon and L and R are its minimum/maximum possible coordinate sums, bounded by $2 \cdot 10^{14}$.

PROBLEM:

You're given an array A. Let Z be the sum of its two largest elements.

FInd out whether there exists a rearrangement of A such that $A_i + A_{i+1} < Z$ for each $1 \le i < N$.

EXPLANATION:

Let M_1 and M_2 be the two largest elements of A, with $M_1 \ge M_2$.
It is enough to check whether there exists a rearrangement of A such that no occurrence of M_1 is adjacent to any occurrence of M_2.

There are two possible cases here: M_1 can equal M_2, or they can be different.

- Suppose $M_1 = M_2$; let this value be M. We want to check whether A can be rearranged such that no two occurrences of M are adjacent.
  The absolute best we can do is to place instances of M at alternate positions, i.e, position $1, 3, 5, 7, \ldots$
  So, the best we can do is $\lceil \frac{N}{2} \rceil$ positions.
  Hence, if the count of M in A exceeds $\lceil \frac{N}{2} \rceil$, the answer is `No`. Otherwise, the answer is `Yes`.
- Suppose $M_1 \neq M_2$. Then, there is exactly one occurrence of M_1 in A, so we need to check if all the occurrences of M_2 can be placed non-adjacent to it.
  Obviously, if every other element is M_2 (i.e M_2 occurs $N - 1$ times) this is impossible.
  Otherwise, it's always possible — for example, place all the occurrences of M_2 at positions $1, 2, 3, \ldots$ and place M_1 at position N.
  So, in this case it's enough to check whether the count of M_2 equals $N - 1$ or not.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

There are N objects in a row; each with positive/negative/neutral charge.
Charged objects spread their charge to adjacent neutral objects every second; however, a neutral object receiving *both* a positive and a negative charge at the same instant remains neutral.

Eventually the system reaches a stable state. Find the number of neutral objects in this state.

EXPLANATION:

Since each object only spreads its charge to its neighbors, any neutral object will only be eventually affected by at most two charged objects: the closest one to its left, and the closest one to its right.

Let's consider two **consecutive** charged objects, say at positions L and R. They (and only they) will affect all the neutral objects between them.
For example, if `S = 00+000-0`, the `+` and `-` are consecutive charged objects here.

Let $d = R - L - 1$ be the number of neutral objects between them.

Now,

- If L and R have the same charge, of course everything in-between will receive the same charge as them and cannot remain neutral.
- When L and R have different charges:
  - If something is *strictly* closer to L than R, it'll receive the charge of L.
  - Similarly, something strictly closer to R than L will receive R's charge.
  - So, the only way something can remain neutral is when it is equidistant from L and R. In such a case, it's easy to see that it'll always remain neutral since it will always receive different charges from each side.
- In particular, the third case above happens if and only if d is odd; since we need a unique middle element.

This gives us a pretty simple solution: for each maximal segment of `0`-s in the string, add 1 to the answer if it has odd length and its endpoints have different charges.

There is one edge case: if the string contains only neutral objects, then of course the answer is N.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Basic algebraic manipulation, frequency arrays

PROBLEM:

Given a permutation P of $\{1, 2, \ldots, N\}$, count the number of pairs (i,j) such that $1 \le i \le j \le N$ and $i+j$ divides $P_i + P_j$

EXPLANATION:

The condition "$i+j$ divides $P_i + P_j$" can be rewritten as "there exists an integer k such that $P_i + P_j = k \cdot (i+j)$".

Notice that the above equation can be rewritten as $P_i - k \cdot i = k \cdot j - P_j = -(P_j - k \cdot j)$.
In particular, if the value of k is fixed, it's not hard to find, in linear time, all pairs (i,j) that achieve divisibility using this k.

How?

Iterate over j from 1 to N. Also maintain a frequency array $freq$, which is initially empty.
When at an index j,

- Add 1 to $freq[P_j - k \cdot j]$
- The number of $i \le j$ that satisfy the condition is then exactly $freq[k \cdot j - P_j]$

This can be done in $\mathrm{O}(N)$ time.

However, repeating the above algorithm for each k is too slow, since k can be anything at all and each run takes $\mathrm{O}(N)$ time.

Let's make a few observations to reduce the work we do.

The first thing to notice is that P is a permutation, which means that $P_i \le N$ for every i.
In particular, $P_i + P_j \le 2N$. Combine this with the fact that $i+j \ge 2$, and you can see that any valid k must satisfy $1 \le k \le N$.
We now have a solution in $\mathrm{O}(N^2)$, which is still too slow.

Now, suppose we fix the value of k. Do we really need to look at every element of the array?

No!

Again, let's look at $P_i + P_j = k \cdot (i+j)$. We know $P_i + P_j \le 2N$, so $k \cdot (i+j) \le 2N$.

This tells us that $i+j \le \frac{2N}{k}$: in particular, $j < \frac{2N}{k}$.

So, we don't need to iterate across the whole array: once k is fixed, only run the initial algorithm on the first $\frac{2N}{k}$ elements.

In fact, this optimization immediately improves our complexity: the number of operations we perform is

$$\sum_{k=1}^{N} \frac{2N}{k}$$

which is $\mathrm{O}(N \log N)$, being a **harmonic summation**.

Note that the constraints in this problem are somewhat high, and so using a `map/unordered_map/dict` to maintain your frequency array will most likely result in a TLE verdict because of the high constant factor and/or an extra $\log$ factor.

Instead, you can maintain a global frequency array of size $4N$ and use an offset to account for negative numbers.
That is, if you want to add 1 to the frequency of x, you instead add 1 to $freq[x + 2N]$.
This shifts the indices of the frequency table from $[-2N, 2N]$ to $[0, 4N]$.

Once you've used the frequency table for a specific k, make sure to reset it properly.
If working with an offset-shifted array like this is new to you, take a look at the code linked below.

TIME COMPLEXITY:

$\mathrm{O}(N \log N)$ per testcase.

PROBLEM:

You have two strings A and B, which you can rearrange as you wish to form A' and B'.
Is it possible to make $A' + B'$ a palindrome?

EXPLANATION:

Suppose $N \leq M$, i.e, A is the shorter string.

Now, suppose we're able to obtain A' and B' such that $A' + B'$ is a palindrome.
Since $N \leq M$, this means that:

- The last N characters of B' must exactly form A'.
- The remaining $M - N$ characters of B' must form a palindrome among themselves.

Notice that this isn't too hard to check:

- Let $\text{freq}_A[c]$ denote the number of occurrences of c in A. Similarly define $\text{freq}_B[c]$.
- The first condition then simply says that $\text{freq}_A[c] \leq \text{freq}_B[c]$ for *every* character c from `'a'` to `'z'`.
  - This can be checked quite easily by just building both frequency tables.
- The second condition requires us to check whether all the remaining characters can form a palindrome via rearrangement.
  - This is a rather classical task, and has a simple solution: a list of characters can be rearranged to form a palindrome if and only if at most one of them occurs an odd number of times.
  - Notice that this is also easy to check with the frequency tables we have: the number of c such that $(\text{freq}_B[c] - \text{freq}_A[c])$ is odd should be ≤ 1.

The answer is affirmative if and only if both conditions above are satisfied.

If $N > M$ just swap A and B and apply the above algorithm anyway.

TIME COMPLEXITY

$\text{O}(N)$ per test case.

PROBLEM:

On a string S, you can perform the following operation:

- Choose an index $1 \le i \le |S|$, and replace S_i with three occurrences of S_i

You're given a string U, find a string S such that it can be converted into U using the maximum number of moves possible.

EXPLANATION:

In this problem, it helps to look at the process in reverse.
That is, instead of replacing a character with three of its occurrences, we take three consecutive occurrences of a character and replace it with a single one.

For example, the string aaaabccc can be turned into aabccc or aaaabc in one move this way.

So, we can start from U and then perform this reverse move as many times as possible to obtain the string S we're looking for.

To do this, observe that we can simply greedily delete characters.
That is, as long as there are three equal adjacent characters, delete two of them.

Directly deleting from a string can be slow, so you'll need to be a bit clever with implementation.
One way to implement this is as follows:

- Let S be the answer string, initially empty.
- Iterate i from 1 to N.
- Append U_i to S, by doing `s += u[i]` (note that `s = s + u[i]` is a linear operation and will lead to TLE in this problem)
- If the last three characters of S are now equal, delete two of them. Since we're deleting from the end, this can be done in $\text{O}(1)$ time.

Finally, print the string S.

TIME COMPLEXITY

$\text{O}(N)$ per test case.

PROBLEM:

Construct an array of length N such that for each $1 \le i < N$,

- If i is odd, $P(i) > S(i)$
- If i is even, $P(i) < S(i)$

$P(i)$ denotes the sum of the first i elements of A, $S(i)$ denotes the sum of the last i elements of A.

EXPLANATION:

Looking at what the inequalities given to us mean in terms of the values of A, we see that:

- $A_1 > A_N$
- $A_1 + A_2 < A_N + A_{N-1}$
- $A_1 + A_2 + A_3 > A_N + A_{N-1} + A_{N-2}$

  $\vdots$

Intuitively, for the inequalities to reverse we see A_2 should be "much smaller" than A_{N-1}; A_3 should be "much larger" than A_{N-2}; and so on.
In general,

- If i is odd, A_i should be much larger than A_{N+1-i}
- If i is even, A_i should be much smaller than A_{N+1-i}

In particular, A_i can never equal $A_{N+1-i} for any $1 \le i \le N$.

Notice that this immediately tells us that when N is odd, no such array can exist: we would require the middle element to not equal itself, which is obviously impossible.

Now let's deal with the case when N is even.

One easy way to deal with the "much larger/much smaller" conditions is to make one of the elements positive and the other negative. Setting one of them to x and the other to $-x$ for an appropriately chosen x will give us what we want.

Let's try to build A in this fashion.

- First, set $A_1 = 1$ and $A_N = -1$. This ensures $A_1 > A_N$.
- Now, if we set $A_2 = -2$ and $A_{N-2} = 2$, we get $P(2) = -1$ and $S(2) = 1$, satisfying the inequality.
- Once again, it's easy to see that setting $A_3 = 2$ and $A_{N-2} = -2$ gives us $P(3) = 1$ and $S(3) = -1$, as necessary.

  $\vdots$

In general, alternating between 2 and -2 (except for the first and last elements) will give us what we want:

- If i is even
  - $P(i) = 1 + (-2) + 2 + (-2) + \ldots + (-2) = -1$
  - $S(i) = -1 + 2 + (-2) + 2 + \ldots + 2 = 1$
- If i is odd
  - $P(i) = 1 + (-2) + 2 + (-2) + \ldots + 2 = 1$
  - $S(i) = -1 + 2 + (-2) + 2 + \ldots + (-2) = -1$

This satisfies all the inequalities we have, and so is a valid array.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Combinatorics

PROBLEM:

Given an increasing array A consisting of integers from 1 to N, count the number of permutations of $\{1, 2, \ldots, N\}$ whose prefix maximums are exactly A.

EXPLANATION:

There's a couple of different ways to solve this, but a proper model of what's going on makes it quite simple.

Let's try to create a permutation P by placing the integers $1, 2, \ldots, N$ in order.

First, note that the first element of P *must* be A_1, since it's the first maximum.
So, we have the following:

- 1 can't be placed at the first position, but anything else is ok since it can't affect the prefix maximums. So, there are $N - 1$ options.
- 2 can't be placed at the first position or where 1 is, but again anywhere else is fine. $N - 2$ options.
- Similarly, 3 has $N - 3$ positions where it can be placed.

  $\vdots$
- $A_1 - 1$ has $N - (A_1 - 1)$ choices.

Once these have been placed, we place A_1 at position 1, as noted earlier.

After this, notice that A_2 *must* be placed in the leftmost empty position for it to be the second maximum.
This fixes $A_1 + 1$ elements, so

- Element $A_1 + 1$ has $N - (A_1 + 1)$ options.
- Element $A_1 + 2$ has $N - (A_1 + 2)$ options.

  $\vdots$
- Element $A_2 - 1$ has $N - (A_2 - 1)$ options.

It's easy to see that this process continues till we place $A_K = N$.

That is, for every integer x that's *not* one of the A_i, it has $(N - x)$ options.

So, the final answer is simply the product of all of these, i.e

$$\prod_{\substack{x=1 \\ x \notin A_i}}^{N} (N - x)$$

which is easily computed in $\mathrm{O}(N)$.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Combinatorics

PROBLEM:

For two binary strings x and y of length N, define $F(x, y)$ as follows:

- For a permutation P of $\{1, 2, \ldots, N\}$, let C and D be defined as:
  - $C_i = x_{P_i}$
  - $D_i = y_{P_i}$
- $F(x, y)$ is the maximum value of $\min(\text{LNDS}(C), \text{LNDS}(D))$ across all permutations P.

Let S denote the set of all binary strings of length N. Compute

$$\sum_{x \in S} \sum_{y \in S} F(x, y)$$

EXPLANATION:

Let's first figure out how to quickly compute $F(x, y)$ for a fixed x and y.
Notice that the values x_i and y_i are essentially 'tied' to each other, i.e, need to be moved to the same final position.
This gives us 4 types of positions, depending on which of x_i and y_i are $0/1$.

Let c_{00} denote the number of positions such that $x_i = y_i = 0$. I'll call these 00-positions.
Similarly define c_{01}, c_{10}, c_{11}.
We need to figure out how to arrange these 4 types optimally to maximize the length of the minimum LNDS.

- It's always optimal to place 00-positions at the start of the array, since they'll contribute to the LNDS of both strings.
- For the same reason, it's optimal to place 11-positions at the end of the array.
  - Together, they ensure the minimum LNDS has length at least $c_{00} + c_{11}$.
- Now we have to deal with the 01- and 10-positions.
  Playing around a little with them, it can be seen that the best we can get is $\max(c_{01}, c_{10}) + \frac{\min(c_{01}, c_{10})}{2}$ (where the division is floor division).

Proof

Our claim is that if we have x 01-positions and y 10-positions, the longest possible minimum LNDS has length $\max(x, y) + \min(x, y)2$.

This will be proved in two parts: a lower bound and an upper bound.

Lower bound

First, let's prove the lower bound by constructing a sequence that has this as its minimum LNDS.
w.l.o.g let $x \geq y$, so $\max(x, y) = x$ and $\min(x, y) = y$.

- First place $y/2$ occurrences of 10.
- Then place all x occurrences of 01.
- Finally, place all remaining occurrences of 10.

So, the first string looks like $\underbrace{111\ldots11}_{y/2}\ \underbrace{00\ldots00}_{x}\ \underbrace{11\ldots11}_{y-y/2}$ and the second looks like $\underbrace{000\ldots00}_{y/2}\ \underbrace{11\ldots11}_{x}\ \underbrace{00\ldots00}_{y-y/2}$

The first $y/2+x$ characters of the first string are non-decreasing, while the last $x+y/2$ characters of the second string are non-decreasing.
This gives us the required lower bound: the minimum LNDS has length at least $x+y/2$.
Upper bound

Now for the upper bound.
Consider a configuration where the first binary string has LNDS of length $> x+y/2$.
In particular, this means that the last $y/2+1$ characters of this LNDS must be all be 1; and everything after them must be 0.
Further, we can also assume that there are no zeros between the ones (if there were, they don't contribute to the LNDS anyway; so we can just move them to the end and it helps out the second string more).

Suppose there are z zeros after these $y/2+1$ positions.
The LNDS of the first string then has length at most $x+y-z$, since the last z positions can't contribute.

Now let's look at the LNDS of the second string.

- If the LNDS doesn't include *any* of these $y/2+1$ positions, then it'll have length strictly less than $x+y/2$ (because there are less than $x+y/2$ remaining positions).
- If the LNDS does include one of them, it might as well include them all: our setup guarantees that the values at these positions are 0 in the second string, and there are no ones inbetween them.
- In particular, the LNDS has length *at most* $y+z$ (where z is as defined above).

We'd like $y+z \geq x+y/2$, which gives us $z \geq x-y/2 \geq y/2$.
In particular, this means the LNDS of the first string has length $\leq x+y-y/2 \leq x+y/2$, a contradiction to us assuming it was strictly greater.

So, the minimum LNDS length cannot exceed $x+y/2$, and our upper bound is proved as well.

With this information in hand, let's compute the answer.
Notice that the score of a pair of strings depends really on only two things:

- How many pairs of same/different positions there are.
- Among the different positions, how many of them are 01 and how many are 10.

Since $N \leq 2000$, we can fix both.
Suppose there are k positions such that $x_i = y_i$; and m of the remaining $N-k$ positions are 01.
Then,

- The LNDS length of such a pair is $k+\max(x,y)+\min(x,y)/2$, where $x=m$ and $y=N-k-m$. Let this be len.
- The number of such pairs can be computed as follows:
  - Fix the set of k equal positions, which can be done in $\binom{N}{k}$ ways.
  - For each equal position, fix whether it's 00 or 11. This is 2^k choices.
  - Of the remaining $N-k$ positions, choose which m of them are 01-positions. This is $\binom{N-k}{m}$ choices.
- Multiply all three values above to obtain the number of pairs of strings, say count.
- Then, the answer increases by $\text{len} \times \text{count}$, since count pairs of strings have an answer of len.

Do this for all k from 0 to N and m from 0 to $N-k$ to obtain the final answer.

For a fixed k and m, we need to compute a couple of binomial coefficients and a power of 2.
The former can be done by precomputing factorials/inverse factorials (or even precomputing all 2000^2 relevant binomials using Pascal's formula), while the latter can be done using binary exponentiation.

TIME COMPLEXITY:

$\mathrm{O}(N^2)$ or $\mathrm{O}(N^2 \log MOD)$ per testcase.

PROBLEM:

Given an array A of length N, it is possible to partition it into two non-empty subsequences such that the product of their sums is odd?

EXPLANATION:

Suppose we were able to split A into two subsequences S_1 and S_2 satisfying the given condition.

Then, note that we want $sum(S_1) \times sum(S_2)$ to be odd, which is only possible when $sum(S_1)$ and $sum(S_2)$ are both odd.
Further, S_1 and S_2 partition A, and so $sum(S_1) + sum(S_2) = sum(A)$.

Since $sum(S_1)$ and $sum(S_2)$ must both be odd, $sum(A)$ must be even.
So, if $sum(A)$ is odd the answer is immediately “No”.
From now on, let’s consider $sum(A)$ to be even.

If A contains only even numbers, then any subsequence also has even sum so splitting into two subsequences with odd sum is impossible.
On the other hand, if A contains an odd number, say x, then we can simply choose $S_1 = \{x\}$ and S_2 to be all the other elements.

S_1 obviously has odd sum, and since $sum(A)$ is even, S_2 also has odd sum and we’re done.

So, the answer is “Yes” if and only if $sum(A)$ is even *and* A contains at least one odd number.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Knowledge of maps/dictionaries OR sorting

PROBLEM:

You're given an array A. Define the function f as

$$f(L,R) = \sum_{i=L}^{R-1} (A_i - A_{i+1})$$

with $f(i,i) = 0$.

Count the number of unstable subarrays, i.e, subarrays (L,R) such that $f(L,R) \neq A_R - A_L$.

EXPLANATION:

Notice that if you expand it out, $f(L,R)$ is a telescoping sum. That is,

$$\begin{aligned} f(L,R) &= \sum_{i=L}^{R-1} (A_i - A_{i+1}) \\ &= (A_L - A_{L+1}) + (A_{L+1} - A_{L+2}) + \ldots + (A_{R-1} - A_R) \\ &= A_L + (-A_{L+1} + A_{L+1}) + (-A_{L+2} + A_{L+2}) + \ldots + (-A_{R-1} + A_{R-1}) - A_R \\ &= A_L - A_R \end{aligned}$$

So, $f(L,R) = A_L - A_R$, which means a subarray is unstable if $A_L - A_R \neq A_R - A_L$.

Let's instead count the number of subarrays that *aren't* unstable, i.e, which satisfy $A_L - A_R = A_R - A_L$. We can then subtract this from the total number of subarrays, which equals $\frac{N \cdot (N+1)}{2}$
Notice that this equality simply translates to $A_L = A_R$.

So, our objective is count the number of subarrays whose endpoints are equal.
This is the same as counting the number of pairs of equal elements in the array!

We do need an algorithm to do this quickly, since $\mathrm{O}(N^2)$ won't cut it.
There are several ways of doing this faster, here's one:

- Let freq be a `map`/`dict`, such that $\mathrm{freq}[x]$ denotes the number of times we've seen x *so far*.
  Initially, freq is empty.
- Iterate i from 1 to N. When processing index i:
  - Increase $\mathrm{freq}[A_i]$ by 1.
  - Then, add $\mathrm{freq}[A_i]$ to the answer, since that's the number of indices to the left of i whose value equals A_i.

Depending on the kind of map used, this is $\mathrm{O}(N \log N)$ or (expected) $\mathrm{O}(N)$ time, which is fast enough.
There are other implementations that don't need maps; for example using sorting.

TIME COMPLEXITY

$\mathrm{O}(N)$ or $\mathrm{O}(N \log N)$ per test case.

PROBLEM:

There's an $N \times N$ chessboard with N bishops on it.
Initially, the bishops are at positions $(1,1),(2,2),(1,3),(2,4),(1,5),\ldots$

You want their final positions to be $(1,1),(2,2),(3,3),(4,4),\ldots$
What's the minimum number of moves needed to achieve this?

EXPLANATION:

If $N \leq 2$, the answer is 0 because the first 2 bishops are already in positions.

For $N > 2$, working out a few small cases on paper might give you the following strategy:

- For each bishop that's initially on $(2,x)$, use one move to bring it to the required diagonal.
- Then, for each bishop that's initially on $(1,x)$, use two moves to bring it to the required diagonal.

This strategy would take $a + 2 \cdot b$ moves, where $a = \lfloor \frac{N}{2} \rfloor - 1$ is the number of bishops initially on the second row (that need to be moved), and $b = \lceil \frac{N}{2} \rceil - 1$ is the number of bishops initially on the first row (that need to be moved).

In fact, this is optimal!

Proof

For $N \leq 2$,, the fact that 0 moves are required is obvious.

For $N > 2$, note that each bishop has a unique cell on the main diagonal that they can reach in one move.
In particular, the bishops on $(2,2k)$ and $(1,2k+1)$ share the same main diagonal cell for any $k \geq 1$.

So, each of these bishops need at least one move, and at least one of them needs another; because if they both used one move they'd end up in the same cell, which is not allowed.

This gives us a lower bound on the number of moves needed, and it's easy to see that our strategy attains this lower bound and is hence optimal.

It's not hard (but also not necessary) to further simplify this into a formula based on N:

- If $N = 1$, the answer is 0.
- If N is odd, the answer is $\dfrac{3N-5}{2}$
- If N is even, the answer is $\dfrac{3N-6}{2}$

TIME COMPLEXITY

$\mathrm{O}(1)$ per test case.

PROBLEM:

You're given a string S containing the characters `4, 0, *` only.
Each `*` can be replaced with either `0` or `4`.

Find the number of times `404` appears as a subsequence of S, across all possible replacements.

EXPLANATION:

There are two different ways to solve this problem: either somewhat directly using math, or using dynamic programming.

Solution 1 (Math)

Let's start with a slow solution first.

Instead of fixing a way of replacing the `*`s and counting the number of `404` subsequences, let's instead fix a `404` subsequence and figure out the number of replacements it occurs in.

So, suppose we fix indices $i < j < k$ to form the `404` subsequence. Then,

- Each of S_i and S_k should be either `4` or `*`.
  Further, if they're `*`, their replacement is unique (since we want them to be `4`).
- Similarly, S_j should be either `0` or `*`, and if it is `*` its replacement is unique.
- If the above conditions are satisfied, then any `*` that's not at positions i, j, k can be freely replaced by either `0` or `4`.
  So, if there are m such indices, the count is 2^m.

Doing this for each (i, j, k) triplet gives us a solution in $\mathrm{O}(N^3)$.

To optimize this, let's try to only fix the position of j, i.e, where the `0` in the `404` subsequence will be.
For this, we can choose any position such that S_j is either `0` or `*`.

Let the remaining number of `*`s be m.

Once `0` is fixed, let's try to fix i and k, i.e, the `4`s.
There are four options here:

- **Case 1:** We can choose a `4` before j and a `4` after j.
  For any such choice, the number of valid replacements is 2^m.
  The number of such choices equals the number of `4`s before j, multiplied by the number of `4`s after j.
- **Case 2:** We can choose a `4` before j and a `*` after j; and replace the `*` with `4`.
  The number of valid replacements is now 2^{m-1}, since we used up one `*`.
  The number of such choices equals the number of `4`s before j, multiplied by the number of `*`s after j.
- **Case 3:** Choose a `*` before j and a `4` after j.
  This is symmetric to case 2, just with 'before' and 'after' swapped.
- **Case 4:** Choose a `*` before j and a `*` after j.
  Here, there are 2^{m-2} valid replacements.

Notice that each case can be calculated in $\mathrm{O}(1)$ time if we know the number of `4`s before/after index j, and the number of `*`s before/after index j.
These counts are easy to find quickly: for example, you can precompute them using prefix or suffix sums; maintain them as you iterate; or even binary search on a list of positions.

In particular, each index can be processed in $\mathrm{O}(1)$ or $\mathrm{O}(\log N)$ time, which is fast enough.

Solution 2 (Dynamic programming)

Let $\text{dp}[i][404]$ denote the number of occurrences of `404` as a subsequence of the first i characters, across all replacements of `*` in them.
Similarly define $\text{dp}[i][40]$ and $\text{dp}[i][4]$.

Then, we have the following transitions for each $1 \le i \le N$.

- First, for each $x \in \{4, 40, 404\}$, set $\text{dp}[i][x] = \text{dp}[i-1][x]$.
  This is because existing instances of each subsequence will continue to exist.
  Now, let's look at new instances of each one we can create using the i-th character.
- If S_i is `4`, then:
  - Add 2^m to $\text{dp}[i][4]$, where m is the number of `*` before index i.
    This is because the `4` at this index will contribute one occurrence for every possible replacement so far, and there are 2^m such replacements.
  - Add $\text{dp}[i-1][40]$ to $\text{dp}[i][404]$.
    This is because we can create a new instance of `404` from a previously existing instance of `40`.
- If S_i is `0`, add $\text{dp}[i-1][4]$ to $\text{dp}[i][40]$.
  This is because we can create a new instance of `40` from an already existing instance of `4`.
- If S_i is `*`, perform both transitions.
  Note that in this case, you should initially set $\text{dp}[i][x] = 2 \cdot \text{dp}[i-1][x]$; once for each replacement.

The final answer is $\text{dp}[N][404]$.

For ease of implementation, the states can be renamed into $\text{dp}[i][0], \text{dp}[i][1], \text{dp}[i][2]$ corresponding to $4, 40, 404$ respectively.

TIME COMPLEXITY

$\text{O}(N)$ per testcase.

PROBLEM:

Given an array A and an integer B, does there exist a subsequence of A whose bitwise AND equals B?

EXPLANATION:

First, recall the following property of bitwise AND:

$$\text{If } x \,\&\, y = z, \text{ then } x \,\&\, z = z \text{ and } y \,\&\, z = z.$$

That is, if $x \,\&\, y = z$, then both x and y will be *supermasks* of z (meaning any bit set in z will also be set in x and y).

This of course extends to the bitwise AND of several values as well:
If $x_1 \,\&\, x_2 \,\&\, \ldots \,\&\, x_k = y$, then each x_i will be a supermask of y.

Let's look at the above property from the perspective of the problem we want to solve.
We want to figure out whether there's a subsequence of A such that its bitwise AND equals B.

That is, we want to know whether there exist indices $1 \le i_1 < i_2 < \ldots < i_k \le N$ such that

$$A_{i_1} \,\&\, A_{i_2} \,\&\, \ldots \,\&\, A_{i_k} = B$$

Immediately, the first property tells us that each A_{i_j} will be a supermask of B.
So, we only care about those values in A that are supermasks of B.

Let S be the set of values in A that are supermasks of B.
Then, in fact S is the subsequence we're looking for!

That is, if the bitwise AND of the values in S equals B, the answer is `Yes`.
Otherwise, the answer is `No`.

Why?

If the bitwise AND of S equals B, obviously the answer is `Yes` since we directly found a valid subsequence.

Now, suppose the bitwise AND of S doesn't equal B.
Since every element of S has B as a submask, clearly their AND must also have B as a submask.
However, since it doesn't equal B, the AND must also be a strict supermask of B — that is, there exists some bit k such that B doesn't have k set; but every element of S does have k set.

Now, consider an arbitrary subsequence of A.

- If this subsequence includes an element outside S, clearly its bitwise AND cannot be B (by definition of S).
- If this subsequence is fully contained in S, then its AND will have bit k set for sure; and so cannot equal B anyway.

So, no subsequence has a bitwise AND of B, as claimed.

This gives a simple linear solution: find the bitwise AND of all supermasks of B, and check if it equals B or not.

TIME COMPLEXITY

$\mathrm{O}(N)$ per testcase.

PROBLEM:

Given the solve counts of Om and Addy on each of N days, find which one among them had the longer streak.

EXPLANATION:

First, let's try to find Om's longest streak of solving problems.
That is, we want to find the longest subarray of A that contains only non-zero elements.

That can be done by iterating across the array and maintaining the current length of a non-zero subarray.
In particular,

- Let cur be a variable denoting the current length of a non-zero subarray.
  Initially, $\text{cur} = 0$.
- Then, for each i from 1 to N:
  - If $A_i > 0$, increase cur by 1.
  - Else, set cur to 0.

The maximum value of cur ever reached is the number we're looking for.
Let this value be mx_A.

Similarly, Addy's maximum streak can be found by applying this algorithm to array B, giving us the value mx_B.

Finally, compare mx_A and mx_B, and print `Om`, `Addy`, or `Draw` depending on which one is higher or if they're equal.

TIME COMPLEXITY

$\text{O}(N)$ per testcase.

PROBLEM:

You're given two integer arrays A and B of length N. In one move you can:

- Pick $1 \le i < j \le N$, and swap one digit of A_i with one digit of A_j. This operation is free.
- Pick $1 \le i \le N$, and replace some digit of A_i with some other digit. This operation has a cost of 1.

Find out whether you can make $A = B$ with a cost of at most K.

EXPLANATION:

First, note that neither operation allows us to change the length of a string.
So, if $len(A_i) \neq len(B_i)$ for some $1 \le i \le N$, then making them equal is obviously impossible.

Now, note that the first operation allows us to freely rearrange the digits in A as we like, so our aim should be to just make sure that A and B have the same multiset of digits.

So, let $\text{ct}_A[d]$ be the number of times digit d appears in A, and $\text{ct}_B[d]$ be the same for B.

Let $S = \sum_{d=0}^{9} \max(0, \text{ct}_A[d] - \text{ct}_B[d])$, i.e, S is the number of 'extra' digits that A has compared to B.
We need one operation to get rid of each of these digits, so we definitely need at least S operations of type 2.

It's also not hard to see that we need exactly S operations of type 2 to make the multisets equal.

Proof

Recall that we ensured that A_i and B_i have equal lengths, right at the start.
So, the total number of digits in A equals the total number of digits in B.

In particular, for each digit that A has extra of, there will be some digit that it has a deficit of.
So, in one replace operation, we can turn an extra digit into one that we require.

This allows us to use exactly S operations so that A and B have the same multisets of digits, as required.

So, once S is computed, the answer is `Yes` if $S \le K$ and `No` otherwise.

TIME COMPLEXITY

$\text{O}(N)$ per test case.

PROBLEM:

There is a strip of N cells, some empty and some blocked.
Alice and Bob play a game, with Alice starting first.
On their turn, a player does the following:

- Choose K contiguous empty cells on the strip, and place dominoes on these K cells.
- A domino placed on turn T will disappear at the start of turn $T + D$.

A player who cannot make a move loses.
Find out the winner with optimal play; or whether the game will go on infinitely.

EXPLANATION:

As a first observation, note that if the game hasn't finished by the end of the D-th turn (under optimal play), it will go on forever.
This is because, on the $(D + i)$-th turn, the dominos placed on the i-th turn will disappear; so the current player will always have a move to make.

This means we can discard the 'disappearing' part entirely, and just work with placing normal dominoes.
The 'normal' game will, of course, have a unique winner — and obviously, this person is the only one who has a chance of winning the disappearing game as well.
We only need to check at the end whether the winner can force a win in $< D$ moves; if they can't, it's a draw.

This also tells us what each player's strategy is.
Suppose Alice wins the normal game. Then,

- Alice will make a move that allows her to force a victory in the minimum number of moves.
- Bob will make a move that allows him to prolong the game as much as possible; in an attempt to reach a draw.

If Bob wins, the roles reverse.

Together, this tells us the two pieces of information we need to know:

- Who the winner of the normal game is; and
- The least number of moves they need to obtain victory.

Both of these can be found using dynamic programming.

Each state of the strip can be represented using N-bit bitmasks.

We will compute the following pieces of information for each bitmask:

- $\text{winner}[mask]$, which is a boolean value.
  A value of 1 or `True` means that the person who plays first on this mask will win it; while a value of 0 or `False` means that the person going second wins.
  A mask will be called a *winning state* if $\text{winner}[mask] = 1$, and a *losing state* otherwise.
- $\text{time}[mask]$, which is the 'optimal' number of turns for this mask for the first player.
  That means the following:
  - If $\text{winner}[mask] = 1$, the $\text{time}[mask]$ is the *minimum* time needed for victory.
  - Otherwise, $\text{time}[mask]$ is the *maximum* amount of time the game can be prolonged for.

These values can be computed as follows:

- If no moves can be made from $mask$, then $\text{winner}[mask] = 0$ and $\text{time}[mask] = 0$.
- Otherwise, let S_{mask} be the set of states that can be reached from $mask$ by making exactly one move.
  If S_{mask} contains any losing state, then $\text{winner}[mask] = 1$; otherwise $\text{winner}[mask] = 0$.

- If $\text{winner}[mask] = 0$, then $\text{time}[mask] = 1 + \max(\text{time}[nxt])$ across all $nxt \in S_{mask}$.
  That is, if the current state is losing, then the player will choose to make a move such that the opponent needs as many possible moves to win.
- If $\text{winner}[mask] = 1$, then $\text{time}[mask] = 1 + \min(\text{time}[nxt])$ across all $nxt \in S_{mask}$, such that $\text{winner}[nxt] = 0$.
  That is, if the current state is winning, the player will choose to make a move such that they can win in as little time as possible; while also ensuring that they move to a losing state before passing the turn.

If the winner and time values are memoized, performing the above computations recursively is enough to solve the problem.

However, a little care is needed with implementation.
In particular, you might notice that there are technically 2^N possible bitmasks; and for $N = 30$ we certainly can't compute the answers for them all quickly enough.

However, in most cases, the set of bitmasks that are actually reachable is generally much smaller because we place multiple dominoes at the same time.
For example, with $N = 30, K = 2$ and a completely empty starting strip, the real number of reachable states is only slightly more than 10^6 which is very manageable.

The only outlier is if $K = 1$, in which case every mask can indeed be reached since we place dominoes one at a time.
However, $K = 1$ can be handled separately: if there are $\geq D$ empty spaces, the game is a draw; otherwise the winner is decided uniquely by the parity of the number of empty cells (Alice if odd, Bob if even).

For $K \geq 2$, recursively processing all reachable bitmasks and using a hashmap for memoization is good enough to get AC.

TIME COMPLEXITY

About 10^7 hashset/hashmap operations, in the worst case.

PREREQUISITES:

Finding next/previous greater element

PROBLEM:

You have an array A. The beauty of a subarray is the difference between its largest and second largest element. Find the number of distinct beauties across all subarrays.

EXPLANATION

We only really care about the maximum and second maximum element of the subarray, so let's fix one of them and try to find valid values for the other one.

In particular, let's fix the *second* maximum of the subarray.
Let's say that we want a subarrays where A_i is the second maximum.

For this, the subarray should definitely contain A_i, and it should certainly contain something else larger than (or at least, equal to) A_i as well.
In particular,

- Let $j < i$ be the highest index such that $A_j \geq A_i$.
- Let $k > i$ be the lowest index such that $A_k \geq A_i$.

Then our subarray should contain at least one of A_j and A_k (giving beauties of $A_j - A_i$ and $A_k - A_i$, respectively)
In fact, these two are the only beauty values we need to care about with A_i being the second maximum!

Proof

Let's consider a few cases:

- Suppose the subarray contains both A_j and A_k. Then, these two elements are $\geq A_i$, so at the very least we can ensure that A_i won't be picked as the second maximum; and such a subarray will be taken care of when considering something else as second maximum.
- Suppose the subarray contains A_j but not A_k. Then, if A_i is the second maximum, the beauty obtained is $A_j - A_i$ (which we do consider); and if A_i is not the second maximum this subarray will be considered when something else is chosen as second maximum.
- The same analysis applies for when it contains A_k but not A_j.

So, we have a total of (at most) $2N$ possible beauties, if we can compute them all quickly enough then counting the number of distinct ones is trivial: for example, put them all in a `set` and find the size of the set.

Notice that all we really need to be able to do is compute the j and k mentioned above quickly enough, for a given i.

Let's focus on computing j first; the exact same method can be used to find k.
This is a rather classical problem, often called the "previous greater element" problem; and we can find it for every i from 1 to N in $\mathrm{O}(N)$ time total using a stack.

How?

Let $\mathrm{prev}[i]$ be the value of j we're looking for.

Keep a stack S, initially empty. Then, do the following:

- Iterate i from 1 to N.
- For a fixed i,

- While S is not empty, if $A_i > A_{S.top}$, pop the stack
- Then, if S is empty set $\text{prev}[i] = -1$; otherwise set $\text{prev}[i] = S.top$
- Finally, push i onto the stack

The idea is that S will always contain a sorted list of elements (decreasing in value from bottom to top). When adding a new element, remove things less than whatever is being added; then the first element on the stack is the nearest greater element.

Since each index is pushed once and popped at most once, this entire algorithm runs in $\text{O}(N)$ time.

You can also read about this method **here**.

Use the above method to compute the next and previous greater elements for each i, then find all $2N$ candidates and count the number of distinct elements among them.

TIME COMPLEXITY:

$\text{O}(N \log N)$ per testcase.

PROBLEM:

There are N stones and K colors. Each stone must be colored with one of the K colors.
However, any two stones with the same color should have a distance of at least K between them.

How many ways exist to color the stones?

EXPLANATION:

Let's try to color the stones one at a time, from 1 to N.

- Stone 1 can be colored in any of the K colors, there are no restrictions yet.
- Stone 2 has $K-1$ options, since it can't be the same color as 1.
- Stone 3 has $K-2$ options, it can't be the same color as the previous two.

  $\vdots$
- Stone $K-1$ has 2 options.
- Stone K has 1 option, since it can't be any of the previous $K-1$ colors (which are all distinct).
- Stone $K+1$ has only one option: it can't take the color of any of stones $2, 3, \ldots, K$; but it can take the color of stone 1.
- Similarly, stone $K+2$ *must* have the same color as stone 2, and so on.

In general, if $i > K$ then stone i will have exactly one option: have the same color as stone $i-K$.

Multiply the choices for each stone to obtain the final answer.
In particular, you might notice that:

- If $K \le N$, the answer is $K!$ (we multiply each of $2, 3, \ldots, K$ exactly once; everything else is 1)
- If $K > N$, the answer is $K \times (K-1) \times \ldots \times (K-N+1) = \frac{K!}{(K-N)!}$

TIME COMPLEXITY

$\mathrm{O}(K)$ per test case.

PROBLEM:

Given a binary string S of length $2N$, partition it into two distinct sequences of length N each, or claim that this isn't possible.

EXPLANATION:

If S consists of only 0's or only 1's, a valid partition obviously doesn't exist.
In every other case, an answer does exist, and one possible construction is given below.

Let c_0 and c_1 denote the number of 0's and 1's in S.

Suppose $c_0 \leq c_1$.
Then, note that $c_1 \geq N$, i.e, there are at least N ones.

Let one subsequence consist of N ones, and the other one consist of all the other characters.
The other subsequence has at least one 0 in it, so these two subsequences are distinct and we are done.

If $c_0 > c_1$ there are at least N zeros, so we can instead choose one subsequence to be all 0's — the same conclusion holds.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

Given N, construct an $N \times N$ boolean matrix such that the $2N$ strings representing its rows and columns are all distinct.

EXPLANATION:

If $N = 2$, it's impossible to construct a valid matrix: this is covered by the sample cases.

For $N \geq 3$, it's always possible to find a valid matrix.
There are several constructions, here's one:

Suppose you had two binary strings A and B. Two simple conditions for when they aren't equal are:

- A and B start with different characters (for example, 1001 and 0001); or
- A and B start with the same character, but the longest equal prefix is of different length (for example, 0001 and 0011).

Using these ideas, a natural starting point is a matrix that looks something like

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 0 & \dots & 0 \\ 1 & 1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

That is, a matrix whose lower triangle and diagonal are entirely filled with ones, and upper triangle with zeros.

You might notice that this matrix *almost* works: the rows are distinct from each other since they start with different numbers of ones; similarly the columns are distinct from each other.
Also, every row starts with a 1 and almost every column starts with a 0, so they're distinct as well.

The only issue is that the first column equals the last row.
All we need to do is find a minor change in one of them that makes them different, while keeping the others distinct.

One way to do this is to change exactly one cell, and set $A_{2,1} = 0$.
This results in the matrix

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 1 & 1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

which satisfies the conditions when $N \geq 3$.

TIME COMPLEXITY

$O(N^2)$ per testcase.

PROBLEM:

Given **even** N, print a permutation of length N such that all the values of $P_i + P_{N+1-i}$ are distinct for $1 \le i \le \frac{N}{2}$

EXPLANATION:

There are many different constructions that will work. For the most part, the simplest way to come up with a construction is to try to make the values of $P_i + P_{N+1-i}$ sorted in some order.

For example, we can do the following:

- Set $P_1 = 1$ and $P_N = 2$
- Set $P_2 = 3$ and $P_{N-1} = 4$
- Set $P_3 = 5$ and $P_{N-2} = 6$

  $\vdots$
- Set $P_i = 2i - 1$ and $P_{N+1-i} = 2i$

It's easy to see that $P_i + P_{N+1-i} = 4i - 1$, which is obviously distinct for each i.

The created permutation is $[1, 3, 5, \ldots, N-1, N, N-2, N-4, \ldots, 6, 4, 2]$.

There are other constructions; for example, $[1, 2, 3, \ldots, \frac{N}{2}, N, N-1, N-2, \ldots, \frac{N}{2}+2, \frac{N}{2}+1]$, which is what the code below implements.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

You are given an array A. In one move, you can pick $i < j$ and set *both* A_i and A_j to $A_i + A_j$.
Find the minimum number of moves to make every element divisible by 3.

EXPLANATION:

Clearly, only the value of each element modulo 3 matters.
So, take every element modulo 3: now we have to deal with only arrays consisting of $0, 1, 2$, and our objective is to make everything 0.

Combining a 1 and a 2 gets us two zeros, which is nice: we'd like to do this as much as possible.
In fact, this is the only way to get zeros from two non-zero numbers, so we'd like to set things up to have as many zeros as possible.
In particular, it'd be nice if the number of 1's and 2's was close to equal.

Note that combining two 1's gets us two 2's, and vice versa.
So, if the difference between the number of 1's and 2's is very large, we can use this to bring their numbers closer together.

In particular, if there are x ones and y twos, then as long as $|x - y| \geq 3$, performing this move appropriately is optimal since it reduces the difference by at least 2.

Now we're left with only $|x - y| \leq 2$. For convenience, let's assume $x \leq y$.
Deal with each case separately.

x = y

Nothing more needs to be done: we simply perform x moves to bring everything to 0.

y = x+1

There are two cases to consider here: $x = 0$ and $x > 0$.

If $x = 0$, we have an array of the form $[2, 0, 0, 0, \ldots, 0]$.
Working this by hand, you can see it takes 6 moves.

If $x > 0$, in particular there is at least one 1 in the array.
Combine a 1 and a 2 till you reach the array $[1, 2, 2, 0, 0, 0, \ldots, 0]$.
Now, instead of forming two more zeros, first convert it to $[1, 2, 2, 1, 0, \ldots, 0]$, which can then be solved in two moves: this is optimal, and uses $x + 2$ moves.

y = x+2

Again, there are a couple of minor special cases to consider.

If $x \geq 2$, then we can bring the array to the state $[1, 2, 2, 2, 0, 0, \ldots, 0]$, then use 2 moves to make it $[1, 2, 2, 2, 1, 1, 0, \ldots, 0]$, after which we need 3 moves move; for a total of $x + 4$ moves.

In fact, the above method only needs one free position (and $x \geq 1$), since we can perform the moves $[1, 2, 2, 2, 0, \ldots] \to [1, 2, 2, 2, 1, \ldots] \to [0, 0, 2, 2, 1, \ldots] \to [0, 1, 2, 2, 1, \ldots]$, which still takes a total of $x + 4$ moves.

The only remaining cases are:

- $x = 0$, when we have $[2, 2, 0, 0, \ldots, 0]$; which needs 5 moves
- $x = 1$ and no free position, i.e, the array $[1, 2, 2, 2]$. This needs 6 moves.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

There are N types of candies, and you have A_i of the i-th type.
Find the maximum number of groups of candies you can make such that:

- Each group contains exactly K candies; and
- Within a group, all candies are distinct

EXPLANATION:

If we can make x groups satisfying the condition, we can certainly make $x - 1$ groups.
This allows us to use binary search to find the largest valid x.

Now, our task switches to the following:
Given x, is it possible to make exactly x groups of candies satisfying the given conditions?

To answer this, let's look at each of the types of candies.
For the i-th type, we have A_i of them. However, we're trying to make x groups, and each group can contain at most one of this type.
So, we can only really use $\min(x, A_i)$ candies of type i.

This means the total number of candies usable by us is $S_x = \sum_{i=1}^{N} \min(x, A_i)$.

Now, since each group should have size K, that means we need $x \cdot K$ candies in total.
So, if $S_x < x \cdot K$, then surely making x groups is impossible.
On the other hand, if $S_x \geq x \cdot K$, it's not very hard to see that making x groups is possible (since we have at most x of each type now).

This gives us a check in $\mathrm{O}(N)$ for the binary search: compute $S_x = \sum_{i=1}^{N} \min(x, A_i)$, and compare it against $x \cdot K$.

TIME COMPLEXITY

$\mathrm{O}(N \log (\sum A_i))$ per testcase.

PROBLEM:

You have N songs saved on your Dotify app. The i-th song is M_i minutes long and in language L_i.
Find the longest playlist you can make if:

- Exactly K songs from these N should be chosen
- Each song should be of language L

EXPLANATION:

We only care about songs whose language is L.
So, let's create a list of the lengths of only these songs, say A.

We now want to choose K elements from A such that their sum is as large as possible.
So,

- If the length of A is less than K, choosing K songs is of course impossible.
  In this case, the answer is `-1`.
- Otherwise, clearly it's optimal to choose the largest K elements of A and find their sum.

A simple way to do this is to sort A and then take the sum of its last K elements.
However, the constraints here are small, so a solution in $\mathrm{O}(N \cdot K)$ that repeats "find the maximum element in A, add it to the sum, and then remove it from A", K times, will also get AC.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ or $\mathrm{O}(N \cdot K)$ per test case.

PREREQUISITES:

Point update/range query structures (**segment trees**/**fenwick trees**), basic combinatorics

PROBLEM:

There are N rocks, the i-th of which is at position X_i and has weight W_i.

Rocky will eat a subset of these rocks starting from the one with least weight, then moving to the one with second least weight, and so on.
Find the expected distance travelled by Rocky if any subset is equally likely to be chosen.

EXPLANATION:

First off, each subset has an equally likely chance of being chosen, and there are 2^N subsets.
So, to find the expected value, it's enough to compute the total distance traveled by Rocky across all subsets, and then divide this by 2^N.

As is common with several problems of this type (i.e, find the sum of something across all subsets), instead of fixing a subset and computing the desired quantity for it, we can fix a quantity and see how many subsets it occurs in.

In our case, the quantity we're interested in the distance between points, so let's look at that.
In particular, we care about distances between *adjacent* (in terms of weight) points in the chosen subset; since the total distance covered is the sum of these distances.

This quickly gives rise to a solution in $\mathrm{O}(N^2)$, as follows:

- Let's fix two indices i and j (without loss of generality, say $W_i < W_j$), and count the number of subsets in which Rocky travels directly from X_i to X_j.
- In order for this direct travel to happen, the chosen subset cannot contain any weights lying strictly between W_i and W_j.
  On the other hand, any weights $< W_i$ or $> W_j$ can be freely chosen.
- This gives us $2^{W_i-1+N-W_j}$ subsets in which this direct travel happens, each for a distance of $|X_i - X_j|$.

With i and j fixed, this is an $\mathrm{O}(1)$ (or $\mathrm{O}(\log MOD)$) computation, and we have our quadratic solution.

To optimize this, let's fix the value of j, and look at the structure of all i that we need to compute for.

For now, only consider $X_i < X_j$; the other side can be computed similarly.

For a fixed i, the contribution is $(X_j - X_i) \cdot 2^{W_i-1+N-W_j}$.
Here, 2^{N-W_j} is a constant independent of i, so let's ignore it for now and multiply it in the end.
The remaining part can be split into $(X_j \cdot 2^{W_i-1})$ and $-(X_i \cdot 2^{W_i-1})$.

We want the sum of this across all i such that $X_i < X_j$ and $W_i < W_j$.
Notice that all we really need is the sum of all 2^{W_i-1} and the sum of $X_i \cdot 2^{W_i-1}$, which are both quantities depending purely on i.

This looks something like a 2D range query, but it can in fact be done with a simple segment tree.

Let's iterate over values of W_j from 1 to N (note that we're iterating from small to large *weight*, not *index*).
Doing this automatically satisfies the constraint $W_i < W_j$ whenever we do a query, so we only need to take care of the $X_i < X_j$ condition.

This means we can simply query for all indices $< j$ (since the input guarantees that $X_1 < X_2 < \ldots < X_N$).
Knowing the 2^{W_i-1} and $X_i \cdot 2^{W_i-1}$ values for these indices, this reduces to a simple range query.

In order to ensure that we don't compute these values for indices with weights $> W_j$, set the values at all indices to be zero initially, and update an index's value only *after* it has been processed as a W_j.
This way, when doing the range sum query, the contribution of higher weight indices is 0, which is what we want.

$X_i > X_j$ can be handled similarly, the range query is on a suffix instead of a prefix.
Don't forget to divide by 2^N in the end.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

Binary search

PROBLEM:

N people want to move up in a building; the i-th person from floor A_i to B_i.
There are M elevators, each one starts at floor 1 and takes 1 second to move up one floor. They cannot move downwards.
A person takes H seconds to enter or exit an elevator.

What's the minimum time needed for everyone to reach their destinations?

EXPLANATION:

The problem screams binary search: after all, if everyone can reach their destinations in $\leq x$ seconds, they can also reach their destinations in $\leq x+1$ seconds.

So, if we were able to check quickly enough (say, in $\mathrm{O}(N)$) whether everyone can reach their destinations in $\leq x$ seconds, we can binary search to find the smallest x for which this is possible.

Let's make a couple of observations to facilitate this check.

Intuition

Note that the time at which the i-th person reaches their destination is completely determined by the last elevator they take; this person will reach at time $B_i - 1 + H \cdot x$, where x is the number of times someone got on/off this elevator before this.

In particular, this means that there exists an optimal solution where each person will only ever use one elevator.

Proof

Consider an optimal solution where a person first uses elevator E_1 and then switches to E_2.
If they had instead started on E_2 and keeps the remainder of their trip the same, they'll still reach the destination at the exact time as before, while E_1 now has 2 less stops and so might reach its destination earlier; it certainly won't reach any later.

So, we've preserved optimality while reducing the number of elevator changes.
Repeat this process to obtain a solution where nobody changes elevators.

Now, let's look at the process from another angle: given an elevator, which set of people do we assign to it?
We've already seen that the only things that matter are the B_i values and the number of stops.

In particular, if we assign k people to an elevator and the highest B_i value among them is M, then the last person to get out will be at time $M - 1 + 2 \cdot H \cdot k$.

The last observation above leads to a rather natural greedy strategy.

First, sort the people in increasing order of B_i. Now, suppose we want to check whether everyone can reach in $\leq x$ seconds.

- First, assign B_N to an elevator. This is guaranteed to be the largest B_i value in this elevator, so we can freely choose upto $k-1$ more people, where $B_N - 1 + 2 \cdot H \cdot k \leq x$.
- Of course, it's best to choose the $k-1$ people with largest remaining B_i values, since that is optimal for the other elevators.
- Then, we can repeat this process on the remaining array. That is, pick the person with largest remaining B_i, then assign as many more people to that elevator as possible while choosing the largest possible B_i

values.

This is rather simple to implement: simply iterate over the B_i in decreasing order while maintaining the current time taken by the elevator.
If adding a new person to it keeps the time $\leq x$, do so. Otherwise, start a new elevator.

This gives us a $\mathrm{O}(N)$ check function, following which the entire problem is solved in $\mathrm{O}(N \log 10^{18})$ using binary search.

Note that the largest possible answer is about $6 \cdot 10^{14}$ so make sure to set the upper limit of the binary search correctly.

TIME COMPLEXITY:

$\mathrm{O}(N \log 10^{18})$ per testcase.

PROBLEM:

An office of 20 workers received N queries. Each worker must solve the same number of queries, and the number of answered queries should be maximum.
How many queries will remain unsolved?

EXPLANATION:

If each person solves x queries, the total number of queries answered will be $20x$, and the unanswered queries will be $N - 20x$.
We want to find the smallest possible value of $N - 20x$ across all x such that $20x \leq N$.

Notice that this is simply the remainder when N is divided by 20.
However, N can be an extremely large number, so we need to deal with it appropriately — in a language like C++, you cannot just print $N\%20$ because N itself is too large to be stored in any inbuilt data type.

Instead, notice that it's enough to keep the last 2 digits of N.
That is, the remainder when N is divided by 20 is the same as the remainder when the last two digits of N are divided by 20 (this is true because 100 is divisible by 20).
For example, $1256\%20 = 16 = 56\%20$.

So, read N as a string, then take only its last two digits and find the remainder when they are divided by 20; that's the answer.

Alternately, you can use a language that does support such large numbers, such as Python or Java.

TIME COMPLEXITY

$\mathrm{O}(|N|)$ per test case.

PREREQUISITES:

Frequency arrays

PROBLEM:

Given an array A, in one move you can pick indices i and j and set A_j to b equal to A_i.
Find the minimum number of moves so that all elements of A become equal.

EXPLANATION:

Suppose we perform some operations and make every element equal to x in the end.
Then,

- x must exist in the original array.
- If A_i was initially equal to x, then no operation is required on this index.
- Otherwise, we need exactly one operation to make A_i equal x.

In particular, this means we need exactly $N - \text{freq}[x]$ operations to make everything equal to x, where $\text{freq}[x]$ is the number of occurrences of x in A.

Our aim is to minimize this, so it's clearly optimal to choose the largest possible value of $\text{freq}[x]$.

This gives us a simple solution: compute the frequency array of A, and let M be its maximum value.
Then, the answer is $N - M$.

TIME COMPLEXITY

$\text{O}(N)$ per test case.

PREREQUISITES:

Basic combinatorics

PROBLEM:

Given two binary strings A and B of length N, count the number of binary strings C of length N such that $\text{dist}(A, C) = \text{dist}(B, C)$.
$\text{dist}(A, C)$ denotes the hamming distance between A and C.

EXPLANATION:

Hamming distance is computed as the sum of N individual terms; one corresponding to each index. So, let's see how each index affects the equality.

Consider an index i.

- If $A_i = B_i$, then this index either contributes 0 to both $\text{dist}(A, C)$ and $\text{dist}(B, C)$ (if $C_i = A_i$); or contributes 1 to both (if $C_i \neq A_i$).
  In other words, it doesn't affect the equality at all, so we can freely choose $C_i = 0$ or $C_i = 1$ here.
- If $A_i \neq B_i$, then depending on our choice of C_i, this index either contributes 1 to $\text{dist}(A, C)$ and 0 to $\text{dist}(B, C)$; or vice versa.
  Clearly, we must choose $\text{dist}(A, C)$ and $\text{dist}(B, C)$ an equal number of times for the equality to hold in the end.

So, suppose there are K positions such that $A_i \neq B_i$; and $N - K$ positions where they're equal.
If K is odd, the answer is immediately 0, since as we noted we need to split these K positions equally. Let's deal with even K now.

At $\frac{K}{2}$ of the unequal positions, we must fix C_i to ensure that $C_i \neq A_i$.
At the other $\frac{K}{2}$ of the unequal positions, we must fix C_i to ensure that $C_i \neq B_i$.
The remaining $N - K$ positions are 'free', and each have two options.

There are $\binom{K}{K/2}$ ways to choose $K/2$ positions out of K, and then the values at these K positions are fixed.
The other positions have 2^{N-K} options in total.
Multiplying everything together, the final answer is

$$\binom{K}{K/2} \times 2^{N-K}$$

Computing binomial coefficients under modulo requires the use of modular division: you can see how **here**.

TIME COMPLEXITY:

$\text{O}(N)$ per testcase.

PROBLEM:

Given N and X, construct a string of length N using the characters +, -, * such that upon evaluation with $N+1$ ones, the result is X.
Multiplication is performed first, addition second, and subtraction last.

EXPLANATION:

Let's think about the maximum and minimum values we can achieve.
Clearly, since there are no parentheses,

- The maximum is $1+1+1+\ldots+1 = N+1$
- The minimum is $1-1-1-\ldots-1 = 1-N$

So, if $X > N+1$ or $X < 1-N$, the answer is immediately -1.
Let's solve for the remaining numbers.

Starting with $1+1+\ldots+1$, notice that replacing one + with a * essentially allows us to 'combine' two ones into a single one, thereby reducing the sum by 1.
For example,

- $1+1+1+1 = 4$
- $1+1+1*1 = 3$
- $1+1*1*1 = 2$
- $1*1*1*1 = 1$

Notice that this allows to create every X between 1 and $N+1$, by using $X-1$ occurrences of + and $N-X$ occurrences of *.

A similar logic applies to the non-positive numbers, where we can start with $1-1-1-\ldots-1$, and replace a - with a * to increase the sum by 1. For example,

- $1-1-1-1 = -2$
- $1-1-1*1 = -1$
- $1-1*1*1 = 0$

In particular, $-X$ can be obtained using $X+1$ occurrences of - and $N-X-1$ occurrences of *.

This means we can obtain every X in the range $[1-N, N+1]$, and so we're done.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

N people order takeout, the i-th person's order will be ready at time X_i and he will arrive to collect it at time Y_i.
$X_i < Y_i$ is known.
Each person will:

- Take his order if it's present.
- If his order is not present but some others are, take whichever present order has the lowest *index*.
- Do nothing if no orders are present.

An *evil person* is one who takes an order that isn't his.
A devil can steal exactly one person's order. What's the maximum possible number of evil people?

EXPLANATION:

The main observation required to solve this problem is the following: at any moment of time, at most one order (that should be present at this time) will be missing.

Proof

Suppose the devil steals person i's order that's supposed to be ready at time X_i; and the person arrives at time Y_i.
Then,

- At any time before X_i, everything is as it should be
- At times in the range $[X_i, Y_i - 1]$, *almost* everything is as it should be: the only difference is that person i's order is missing.
- At time Y_i:
  - If there are no orders present, nothing happens and person i leaves.
    Note that if the devil hadn't stolen his order, then in the normal process there would be no orders remaining at the end of this time anyway; which means the process will go on as usual after Y_i and no more orders will be missing.
  - If there is an order present, person i will take it. Suppose he takes order j.
    Then, at times in the range $[Y_i + 1, Y_j - 1]$, person j's order will be the only one missing (everything else is accounted for, and person i's order would normally have been taken at time Y_i anyway)
- Continue this process to see that at any instant, at most one order is missing.

Note that in the last case in the above proof, the process during time $[Y_i + 1, Y_j - 1]$ proceeds exactly the same as if the devil had stolen person j's order initially!
This observation leads us to a solution.

Let $\text{take}[i]$ denote the index of the takeout that person i will take, if his own is missing (and $\text{take}[i] = -1$ if no such person exists).
Note that since at most one order is missing at any time, $\text{take}[i]$ is uniquely defined.

Now, if the devil steals person i's takeout, it's easy to see that the people who become angry form a chain, going as $i \to \text{take}[i] \to \text{take}[\text{take}[i]] \to \ldots$, as long as the next person exists (i.e doesn't equal -1).

This tells us that we need to compute two things: the $\text{take}[i]$ values for each i, and then the longest chain in them.

Computing take

Let's attempt to find $\text{take}[i]$.
If person i's takeout is missing, then at time Y_i we know that all remaining takeouts will be of persons j such that $X_j < Y_i < Y_j$, i.e, takeouts that were prepared before Y_i but weren't taken yet.

We want to find the smallest j such that this condition is satisfied, and do so for *every* i.
If no such j exists, then $\text{take}[i] = -1$ of course.

Note that the problem constraints guarantee that all the X_i and Y_j are distinct between 1 and $2N$, which means something (in fact, exactly one thing) happens at each time instant.
Let's use that fact to simulate the process nicely:

- Iterate across time, from 1 to $2N$. Also maintain a set of takeouts that haven't been taken yet, say S.
- At time t,
  - If there's an i such that $X_i = t$, insert i into S.
  - If there's an i such that $Y_i = t$, delete i from S.
    Then, (if S is non-empty) $\text{take}[i]$ will be the smallest element of S.

At each time instant, we do only a couple of operations at most.
So, if S were a data structure that allowed us to quickly insert and erase elements, as well as get the smallest element, we'd be done.

There is indeed such a data structure: a `set` (`std::set` in C++, `TreeSet` in Java, Python unfortunately doesn't have one built-in).
This allows us to insert/delete elements in $\text{O}(\log N)$, and get the minimum element in $\text{O}(1)$.

So, make S a set and simulate the process as mentioned above, for $\text{O}(N \log N)$ runtime.
Finding the longest chain

Once the $\text{take}[i]$ values are known, finding the longest chain is fairly straightforward.

Let $\text{dp}[i]$ be the length of the longest chain if the devil steals person i's takeout.
Then,

- If $\text{take}[i] = -1$, we have $\text{dp}[i] = 0$
- Otherwise, $\text{take}[i] = 1 + \text{dp}[\text{take}[i]]$

As long as memoization is used, computing this for every i takes $\text{O}(N)$ time.
The answer is just the maximum of the dp array.

TIME COMPLEXITY

$\text{O}(N \log N)$ per test case.

PROBLEM:

Given X and Y, check whether

$$X^4 + 4Y^2 = 4X^2Y$$

is true.

EXPLANATION:

X and Y are pretty large: so large that you might not be able to compute and store X^4 or $2Y^2$ accurately in some languages (mainly, C/C++).

Instead, notice that

$$\begin{aligned} & X^4 + 4Y^2 = 4X^2Y \\ \iff & X^4 + 4Y^2 - 4X^2Y = 0 \\ \iff & (X^2 - 2Y)^2 = 0 \\ \iff & X^2 = 2Y \end{aligned}$$

That is, the given equation is true if and only if $X^2 = 2Y$, so it suffices to check for that.
Since $X \leq 10^9$ and $Y \leq 10^{18}$, this check does fit within the limts for `long long` and so can be done to solve the problem!

Of course, if you use a language like Python that has support for larger numbers, you can directly check if the initial equation is true or not.

TIME COMPLEXITY

$\mathrm{O}(1)$ per test case.

PROBLEM:

Given integers A, B, C, does there exist an X such that $A \oplus X + B \oplus X = C \oplus X$?

EXPLANATION:

First, let's try to solve the problem for a single bit, i.e, A, B, C can all be either zero or one. We will later generalize this.

Solving for one bit

There are only 8 cases possible here, depending on the values of A, B, C. Trying them out a bit, you might notice that it is in fact almost always possible to find an X that is either 0 or 1 such that $A \oplus X + B \oplus X = C \oplus X$.

There are only two cases where it is impossible:

- $A = B = 0$ and $C = 1$; and
- $A = B = 1$ and $C = 0$

In both cases, choosing $X = 0$ or $X = 1$ won't quite satisfy the equation: you'll end up with either
However, note that choosing $X = 1$ for the first equation and $X = 0$ for the second does somewhat satisfy them, as long as you allow for a 'carry bit'.

That is, you'll end up with the 'equation' $1 + 1 = 0$, but this can still be satisfied if you manage to get an extra of the next higher bit, since it'll then come out to $1 + 1 = 2$.

Solving the general case

Let's try to generalize the above approach.
We saw that trying to satisfy one bit can sometimes depend on how we're able to satisfy the next higher bit.

So, let's iterate across bits in order from lower to higher, i.e, from 0 to 27. Suppose we're at the i-th bit, and the respective values at this bit are A_i, B_i, C_i.
We also may or may not have a carry from the previous bit.

There are two cases now:

- Suppose we don't have a carry. Then, this is the same as the one-bit case, i.e,
  - If $A_i = B_i \neq C_i$, then we can satisfy this bit but the next one will have a carry.
  - In every other case, we can satisfy this bit and the next one won't have a carry.
- Suppose we did have a carry. You can once again do some case analysis and see that we have a very similar-looking result:
  - If $A_i = B_i \neq C_i$, then we can satisfy this bit but the next one *won't* have a carry.
  - In every other case, the next bit *will* have a carry.

This way, we can surely satisfy at least the bits 0 to 27. Now, let's look at where we stand after this process:

- If there is no carry now, we're done! $A \oplus X + B \oplus X = C \oplus X$ has been achieved without any excess remaining, so the answer is "Yes".
- If there is a carry, it's not hard to see that the answer is "No".

So, repeat the process detailed above, and finally check whether there is a carry or not. This is 27 steps per test case.

TIME COMPLEXITY:

$\mathrm{O}(\log \max(A, B, C))$ per testcase.

PREREQUISITES:

Linearity of expectation, Modular multiplicative inverse, basic combinatorics

PROBLEM:

There are A ones and B zeros.
Alice and Bob alternate turns; with Alice moving first.
On their turn, a player chooses (uniformly at random) one of the remaining elements, adds it to their score, and then discards that value.

What is Alice's expected final score?

EXPLANATION:

If you're familiar with linearity of expectation, this is a rather straightforward task.
Each 1 contributes independently to the answer, so let's find the probability that Alice chooses a specific 1 and then multiply this by A: this will be the final answer.

Note that Alice will choose exactly $k = \lceil \frac{N}{2} \rceil$ of the $N = A + B$ elements available. Since each choice is fully random, each set of size k is equally likely to be chosen.
So, there are $\binom{N}{k}$ choices of what the final set can be.

Of these, we'd like to count the number of sets that include a specific 1.
This is not hard: if a 1 is fixed, the other $k - 1$ elements of Alice's set must be chosen from the remaining $N - 1$ elements, giving us $\binom{N-1}{k-1}$ possible choices; again, each one is equally likely.

So, the required probability is $\frac{\binom{N-1}{k-1}}{\binom{N}{k}}$.
Expanding this in terms of factorials and cancelling out will reduce this to just $\frac{k}{N}$.

The final answer is thus simply $A \cdot \frac{k}{N}$, where $k = \lceil \frac{N}{2} \rceil$.
This can be computed in $\mathrm{O}(\log MOD)$, since all that needs to be done is to invert N with respect to MOD.
If you don't know how to compute inverses, a tutorial is linked in the prerequisites above.

TIME COMPLEXITY:

$\mathrm{O}(\log MOD)$ per testcase.

PROBLEM:

There is an $N \times N$ grid, and Chef wants to move from position (x_1, y_1) to (x_2, y_2) using horizontal and/or vertical moves.
Each lateral step has a cost of 1 if either the starting or ending square lie within the grid, and 0 otherwise.

What is the minimum cost to reach (x_2, y_2) from (x_1, y_1)?

EXPLANATION:

There are two cases to consider: Chef can either travel fully within the grid, or he can choose to travel outside of it.

Within the grid

Chef needs to make $|x_2 - x_1|$ horizontal steps and $|y_2 - y_1|$ vertical steps to reach (x_2, y_2), so the minimum cost in this case is their sum, i.e,

$$|x_2 - x_1| + |y_2 - y_1|$$

Outside the grid

When choosing to travel outside the grid, Chef's optimal route is to leave the grid as soon as possible, travel outside of it for some time, then enter it again as close to (x_2, y_2) as possible.

From a point (x, y), the smallest number of moves to reach the border is one out of:

- x (move straight up)
- y (move straight left)
- $N + 1 - x$ (move straight down)
- $N + 1 - y$ (move straight right)

In particular, it is the minimum of these four values.
So, the answer in this case is

$$\min(x_1, y_1, N + 1 - x_1, N + 1 - y_1) + \min(x_2, y_2, N + 1 - x_2, N + 1 - y_2)$$

Consider both cases, compute their answers, and take their minimum as the final answer.

TIME COMPLEXITY:

$\mathrm{O}(1)$ per testcase.

PREREQUISITES:

Linearity of expectation

PROBLEM:

Given N and p, consider the following N-step process:

- Initialize your score S to 0.
- In the i-th step, with probability 50%, add p^{1-i} to S

Let S_i denote your score after i steps.
Find $\mathrm{E}[S_i^2]$ for each $1 \le i \le N$.

EXPLANATION:

Let's first solve this for a fixed k, i.e, compute $\mathrm{E}[S_k^2]$.

We have the set $\{1, p^{-1}, p^{-2}, \ldots, p^{1-k}\}$. We choose a random subset of this set (each with probability 2^{-k} of being chosen), and we want to find the expected value of the square of the sum of this subset.

Suppose the subset we choose is $\{x_1, x_2, \ldots, x_m\}$.
The square of its sum is $(x_1 + x_2 + \ldots + x_m)^2$, which when expanded is

$$\sum_{i=1}^{m} x_i^2 + \sum_{i=1}^{m} \sum_{j=i+1}^{m} 2x_i x_j$$

Let's call the first summation X and the second one Y.
That is, $X = \sum_{i=1}^{m} x_i^2$ and $Y = \sum_{i=1}^{m} \sum_{j=i+1}^{m} 2x_i x_j$.

By linearity of expectation, $\mathrm{E}[S_k^2] = \mathrm{E}[X] + \mathrm{E}[Y]$, so let's compute each one individually.

Computing E(X)

$\mathrm{E}[X]$ is simply the expected sum of squares of the subset.

Once again, let's use linearity of expectation: each element of the set has a 50% chance to be in our subset, so the expected value of the sum of squares is

$$0.5 \cdot (1^2 + (p^{-1})^2 + (p^{-2})^2 + \ldots + (p^{1-k})^2)$$

Notice that this is the sum of k terms of a geometric series, with first term 1 and common ratio p^{-2}.

This sum can thus be computed in $\mathrm{O}(\log MOD)$ using the **sum of GP formula**.

Alternately, you can just maintain the sum of squares as you iterate k, each time increasing it by p^{2-2k}.

Computing E(Y)

For the third this this problem, let's use linearity of expectation.

For each $0 \le i < j < k$, the product $2p^{-i}p^{-j}$ has a 25% chance of occurring in our score (we need to pick both p^{-i} and p^{-j}, 50% chance of each).

So, the required expected value is

$$0.25 \sum_{i=0}^{k-1} \sum_{j=i+1}^{k-1} 2p^{-i-j}$$

For a fixed i, notice that the inner summation is once again a geometric progression with first term p^{-2i-1} and common ratio p^{-1}, so it can be computed in $\mathrm{O}(\log MOD)$.

This already gives us a way to compute $\mathrm{E}[Y]$ in $\mathrm{O}(k \log MOD)$; however, doing this separately for each k is too slow.

One way to speed it up is to flip the order of summations, and look at $\sum_{j=0}^{k-1} \sum_{i=0}^{j-1} 2p^{-i-j}$ instead.

Once again, for a fixed j the inner summation is a GP and can be computed quickly.
However, the nice thing about looking at it this way is that when moving from k to $k+1$, almost all the computations are the same!
In fact, we only need to add in the summation for one more value of j.

This allows us to move from k to $k+1$ in $\mathrm{O}(\log MOD)$ time, since we compute only one sum of GP at each step; hence making the overall complexity $\mathrm{O}(N \log MOD)$.

Even simpler, you can notice that this sum of products increases by p^{1-k} times the sum of all the powers before, so maintaining the sum so far allows this to be updated in $\mathrm{O}(1)$ without dealing with geometric progressions.

There are also other ways to compute this part quickly: for example, the closed-form formula

$$\frac{(p^{k-1}-1)(p^k-1)p^{1-2(k-1)}}{(p+1)(p-1)^2}$$

Either way, $\mathrm{E}[X]$ and $\mathrm{E}[Y]$ can be computed in $\mathrm{O}(\log MOD)$ for a fixed k, so do it separately for each k, add them up, and we're done.

TIME COMPLEXITY:

$\mathrm{O}(N \log MOD)$ per testcase.

PROBLEM:

Given the number of likes and comments on N posts, find out which one is most popular.

EXPLANATION:

This task is mostly an exercise in correctly and directly implementing what is asked for.
There are several ways to solve it, here's one.

Let's maintain the answer index, ans. Initialize ans to 1.

Now, for each i from 2 to N,

- If $A_i > A_{\text{ans}}$, set $\text{ans} = i$ because the i-th post has strictly more likes than the best so far; so it becomes the best so far.
- If $A_i < A_{\text{ans}}$, the i-th post can never be the most popular, so we can ignore it.
- Otherwise, we have $A_i = A_{\text{ans}}$.
  - In this case, compare B_i with B_{ans}. If $B_i > B_{\text{ans}}$, then set $\text{ans} = i$; otherwise don't update ans.

Finally, print ans.

TIME COMPLEXITY:

$\text{O}(N)$ per testcase.

PREREQUISITES:

The process of **Huffman coding**

PROBLEM:

Alice and Bob play a game, independently, on an array of N elements. At each step, a player will take two elements x and y from the remaining set, delete them, and add $x+y$ to the set.
They also add $x+y$ to their score.

Alice has a well-defined strategy: (except for the very first move) she will always use the newly combined element and combine it again.
Bob, meanwhile, can choose his moves as he likes.

Bob wins if his score is strictly less than Alice's.
To achieve this, Bob can change elements of the array as he likes, before any moves are made.
Find the minimum number of changes so that Bob, with optimal play, can guarantee a win over Alice. Also find one such array.

EXPLANATION:

The first step to solving this problem is analyzing Alice's and Bob's best strategies, and seeing when exactly Bob can beat Alice.

Alice's strategy

Let $A_1 \le A_2 \le \ldots \le A_N$.

Alice's best strategy is then as follows:

- Merge A_1 and A_2
- Merge $A_1 + A_2$ and A_3
- Merge $A_1 + A_2 + A_3$ and A_4

$\vdots$

Alice's best score is thus the sum of all prefix sums of (sorted) A, minus A_1.

Bob's strategy

Bob is free to make moves as he likes, so the optimal strategy is to always pick the two smallest remaining elements, Huffman-style. A proof of this optimality can be found **here** .

Now that we know their strategies, it's also not hard to see when exactly Bob can beat Alice: there should be a move when Alice *doesn't choose* the smallest two elements.
That is, in the sorted array A, there should exist an i such that $A_1 + A_2 + \ldots + A_i$ is *not* among the two smallest elements; with the other elements being $[A_{i+1}, A_{i+2}, \ldots, A_N]$.

In particular, notice that this can only happen when $A_1 + \ldots + A_i > A_{i+2}$.

So, Bob's aim is to reach an array where this is the case.

This leads us to a solution that deals with a few cases:

- If $N \le 3$, Bob can never win.
- First, if this condition is already satisfied, nothing more needs to be done — Bob already wins.
- Otherwise, we need to see what we can modify.

First, let's check if one modification is enough.
If i is fixed, we want to see if $A_1 + \ldots + A_i > A_{i+2}$.

The best we can do is bring A_{i+2} down to A_{i+1} (since sorted order needs to be maintained), so if $A_1 + \ldots + A_i > A_{i+1}$, we're done: change A_{i+2} to A_{i+1}.

If this is not satisfied for any $1 \le i < N - 1$, it's always possible to achieve our aim in 2 moves by setting $A_2 = A_3 = A_4$, at which point $A_1 + A_2 > A_4$.

Note that we sorted A to see which changes need to be made, but the actual changes need to be made on the original positions of A; so make sure to keep track of indices as well.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PROBLEM:

Given integers A, B, C, D such that $A \bmod B = C \bmod D$, find the smallest positive integer X such that $(A + X) \bmod B = (C + X) \bmod D$

EXPLANATION:

tl;dr

The answer is either 1 or $\text{lcm}(B, D) - (A \bmod C)$.

First, notice that the actual values of A and C don't matter: only their remainders when divided by B and D, respectively.
So, let's replace A with $(A \bmod B)$ and C with $(C \bmod D)$. In particular, this ensures that $A < B$, $C < D$, and $A = C$.

Now, note that a lot of the time we can just choose $X = 1$.
In fact, as long as $A + 1 < B$ and $C + 1 < D$, we can choose $X = 1$ (since $A = C$, $A + 1 = C + 1$ and both these values don't exceed their respective modulos).
$X = 1$ also works when *both* $A + 1 = B$ and $C + 1 = D$ hold, since both values become 0 in that case.

So, we only need to solve for the case when $X = 1$ *doesn't* work, i.e, exactly one of $A = B - 1$ or $C = D - 1$ holds.

Let's do a bit of math.
Consider an integer X for which equality holds.
We have $(A + X) \bmod B = (C + X) \bmod D$.

Writing out this in terms of remainders, there exist three integers p, q, r such that

- $A + X = pB + r$
- $C + X = qD + r$
  Note that r is common.

It's clearly optimal to choose $r = 0$ (if it was anything higher, $X - 1$ would work too so X wouldn't be minimal).

This gives us $A + X = pB$ and $C + X = qD$.
However, notice that we have $A = C$ (from our replacement in the very first step), so this means $A + X = pB = qD$.

In particular, $A + X$ must be a multiple of both B and D.
The smallest such integer is, by definition, $\text{lcm}(B, D)$.

So, we can choose $X + A = \text{lcm}(B, D)$, which gives us $X = \text{lcm}(B, D) - A$ as our answer.

TIME COMPLEXITY

$O(\log \min(B, D))$ per test case.

PROBLEM:

Given N and M, find the sum of $(\sum_{i=1}^{N} |A_i - B_i|)^2$ across all pairs of arrays A and B of length M such that:

- $0 \le A_i, B_i \le M$
- $sum(A_i) = sum(B_i) = M$

EXPLANATION:

First, let's analyze the value contributed by a single array.
$|A_i - B_i|$ is somewhat hard to deal with, so let's rewrite it as $\max(A_i, B_i) - \min(A_i, B_i)$; after all, that's what their difference is.

Then, notice that

$$\begin{aligned}\sum_{i=1}^{N} |A_i - B_i| &= \sum_{i=1}^{N} (\max(A_i, B_i) - \min(A_i, B_i)) \\ &= \sum_{i=1}^{N} (\max(A_i, B_i) + \min(A_i, B_i)) - 2 \cdot \sum_{i=1}^{N} \min(A_i, B_i) \\ &= 2M - 2 \cdot \sum_{i=1}^{N} \min(A_i, B_i)\end{aligned}$$

So, the sum of differences is fixed if we fix the sum of $\min(A_i, B_i)$ across al indices.
Also, notice that $\sum_{i=1}^{N} \min(A_i, B_i)$ must lie between 0 and M, so there aren't too many possible values for it.

Let's fix $S = \sum_{i=1}^{N} \min(A_i, B_i)$, and try to count how many pairs of arrays A and B attain this value of S.
Each such array will contribute $(2M - 2S)^2$ to the final answer, so if we can quickly count them for a fixed S, we'll be done.

This counting can be done in several steps, as follows:

- First, let's fix the distribution of minimum values across all N indices.
  This is equivalent to saying we have N non-negative integers that sum up to S; and we know there are $\binom{N+S-1}{S}$ such sequences of integers.
- Once this is fixed, we need to distribute a total of $M - S$ more to each of A and B so that their respective sums reach M.
  However, we've already fixed the values of the minimums, so we need to distribute these values only to the maximums; in particular, we can't increase *both* A_i and B_i now, but only at most one of them.
- To account for this, we have the following:
  - Fix the number of positions such that A_i is *strictly* greater than B_i; suppose there are x such positions.
    Note that x can vary from 0 to N.
  - Once x is fixed, also fix which x positions these are: $\binom{N}{x}$ choices in total.
  - After this, we need to distribute $M - S$ to the A_i values of these x positions such that all of them receive at least 1, and we need to distribute $M - S$ to the B_i values of the other $N - x$ positions such all of them receive ≥ 0.
    Both of these can be found as pretty much direct applications of stars-and-bars, giving us the product of two binomial coefficients.

Note that after S and x are fixed, we only compute a few binomial coefficients and multiply them out. There are $M+1$ possible values of S and $N+1$ of x, so this solution is $\mathrm{O}(N \cdot M)$ in total.

tl;dr the answer is

$$\sum_{S=0}^{M} \sum_{x=0}^{N} ((2M-2S)^2 \cdot \binom{N+S-1}{S}\binom{N}{x}\binom{M-S-1}{x-1}\binom{N-x+M-S-1}{M-S})$$

TIME COMPLEXITY

$\mathrm{O}(N \cdot M)$ per test case.

PROBLEM:

Given N, find for each i from 1 to N:

$$\sum_{j=1}^{N} \binom{j-1}{j-\lfloor\frac{j-1}{i}\rfloor-1}$$

EXPLANATION:

There are a couple of different ways to approach this task; one more visual in nature and the other one more algebraic.
They both end up at the same solution, however.

Visual approach

Let's look at what's going on in the context of Pascal's triangle.
For a fixed i:

- Let (x, y) be our position on the triangle. Initially, $(x, y) = (0, 0)$.
- Then, for each j from 1 to N:
  - If j is divisible by i, we move to $(x+1, y)$.
  - Otherwise, we move to $(x+1, y+1)$.

The answer for i is the sum of values at all points we visited.

Looking at the triangle, our path can be broken into several smaller 'straight' paths along diagonals.
In particular, each multiple of i is when we move from one diagonal to another.
So, there are $\lceil\frac{N}{i}\rceil$ such movements in total.
Summing this across all i, we have $\mathrm{O}(N \log N)$ movements in total; which also means $\mathrm{O}(N \log N)$ diagonal segments in total.

Finally, the sum of each diagonal segment can be found in $\mathrm{O}(1)$ time using the **hockey-stick identity**, and so we're done!

Algebraic approach

Note that $\binom{j-1}{j-\lfloor\frac{j-1}{i}\rfloor-1} = \binom{j-1}{\lfloor\frac{j-1}{i}\rfloor}$, because $\binom{N}{K} = \binom{N}{N-K}$.

Let's fix i. Then, notice that if we also fix $x = \lfloor\frac{j-1}{i}\rfloor$, we obtain a certain range of j that attain this value of x; in particular, $x \cdot i \le j-1 < (x+1) \cdot i$.

Let this range be $[L, R]$. Then, we want to compute $\sum_{k=L}^{R} \binom{k}{x}$.
This can be done using the **hockey-stick identity**, and equals $\binom{R+1}{x+1} - \binom{L}{x+1}$.

Now, for a fixed i, there are only $\lceil\frac{N}{i}\rceil$ ranges of j we care about; we don't care about any x such that $i \cdot x > N$.

Since the sum of $\frac{N}{i}$ is $\mathrm{O}(N \log N)$, we compute $\mathrm{O}(N \log N)$ binomial coefficients in total; each of which can be done in $\mathrm{O}(1)$ time.
This is enough to solve the problem.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per testcase.

PROBLEM:

You are given an array B. Find the lexicographically smallest array A such that B_i equals the frequency of A_i for every i; or report that no such array exists.

EXPLANATION:

As with most tasks that require lexicographic minimization, this one can be solved greedily.

Let's try to place the 1's, then the 2's, then the 3's, and so on.

Note that we can always set $A_1 = 1$.
Now, suppose $B_1 = x$. Then, there are exactly $x - 1$ more 1's in A, and to minimize the final array, we must place them at the leftmost $x - 1$ positions j such that $B_j = x$.

Then, we can find the first position > 1 that is unfilled, place a 2 there, and repeat the same process.
Note that at any step if we are unable to find enough positions to place elements, the answer is immediately -1.
In particular, any x must occur $k \cdot x$ times in B (for some $k \geq 0$) for the answer to not be -1.

All that needs to be done is to implement this quickly.

Here's one simple way to do this.
First, build the frequency table of B; let this be $freq$.
If $freq[x]$ is not a multiple of x for some x, the answer is immediately -1; otherwise we'll construct a valid array.

Also keep:

- A variable nxt. Initially, $nxt = 1$. This denotes the next smallest we are going to use.
- A second map, say cur, where $cur[x]$ denotes the current value we're placing that has frequency x.

Let's iterate i from 1 to N. Suppose $A_i = x$. Then,

- If $freq[x]$ is a multiple of x, we must start placing a new element. So, set $cur[x] = nxt$ then increase nxt by 1.
- Then, set $A_i = cur[x]$ and decrement $freq[x]$ by 1.

It's easy to see that the array constructed this way is the same one we discussed initially, since we increase $cur[x]$ after placing exactly x instances of the previous value; and this is of course optimal.

TIME COMPLEXITY:

$\mathrm{O}(N)$ or $\mathrm{O}(N \log N)$ per testcase.

PREREQUISITES:

Longest increasing subsequence

PROBLEM:

There's an $N \times M$ grid with K special cells.
From a cell (x, y):

- If it's a special cell, Chef can move to any one of the 8 neighboring cells
- Otherwise, Chef can move to one of the 4 horizontally/vertically adjacent cells

Find the minimum number of moves to reach (N, M) from $(1, 1)$.

EXPLANATION:

Clearly, in an optimal sequence of moves every move we make is going to be one of the following:

- $(x, y) \to (x, y+1)$
- $(x, y) \to (x+1, y)$
- $(x, y) \to (x+1, y+1)$ (only possible from a special cell)

We need to make $N-1$ moves along x and $M-1$ moves along y, and a special cell allows us to at best combine two of these moves into one (thus saving one step). Going in the 'wrong direction' doesn't decrease the number of steps we need to make, so it's suboptimal.

This also means that any special cells of the form (x, M) or (N, y) don't matter at all, since we can't use them to make the third type of move. Let's discard such special cells first.

Now, notice that if we touch s special cells along our path, the number of moves will be exactly $N + M - 2 - s$. This follows from our observation above: we make $N - 1 + M - 1$ moves, and each special cell allows us to combine two of these into one.

To minimize this, we'd like to find the largest possible value of s, i.e, the largest number of special cells that we can visit along our path.
Note that if (x_1, y_1) and (x_2, y_2) are two special cells, we can visit both of them along our optimal path if and only if $x_1 < x_2$ *and* $y_1 < y_2$.

In particular, if we sort the special cells in ascending order of their x-coordinates, the y-coordinates must form an increasing sequence.
In other words, what we're looking for is just the longest increasing subsequence of the y-coordinates!
This can be computed in $\mathrm{O}(K \log K)$ time in a variety of ways: see here for a couple of them.

Note that there is one extra condition here: we need to ensure that we also pick distinct x-coordinates.
One simple way to ensure this is to sort the cells in increasing order of x, and break ties by *decreasing* order of y.

TIME COMPLEXITY:

$\mathrm{O}(K \log K)$ per testcase.

PREREQUISITES:

Observation, computing binomial coefficients

PROBLEM:

An array B is called good if B_i is either the maximum or the minimum of $B[1 \dots i]$ for each i.

Given an array A, count the number of its rearrangements that are good.

EXPLANATION:

First, let's sort A since it won't change the answer. From now on, $A_i \le A_{i+1}$.

Let B be a good rearrangement of A. Let's analyze the structure of B.

Let $S_L = [B_{i_1}, B_{i_2}, \dots, B_{i_r}]$ be the subsequence of elements that are prefix minimums, and $S_H = [B_{j_1}, B_{j_2}, \dots, B_{j_s}]$ be the subsequence of elements that are prefix maximums. If an element can be both (for example, B_1) then we treat it as a minimum.
S_L and S_H partition B, so $r + s = N$.

For example, if $B = [2, 3, 3, 1, 4, 1]$ then $S_L = [2, 1, 1]$ and $S_H = [3, 3, 4]$.

Note that the following conditions must hold:

- $B_{i_1} \ge B_{i_2} \ge \dots \ge B_{i_r}$ (the minimums decrease)
- $B_{j_1} \le B_{j_2} \le \dots \le B_{j_s}$ (the maximums increase)
- $B_{j_1} \ge B_{i_1}$

Notice that this is an ordering that encompasses every element!
In particular, this means that S_L contains the smallest r elements of A, and S_H contains the largest s elements of A.

So, let's try to fix the value of r and see how many rearrangements exist such that S_L contains the smallest r elements.

Of course, the first element must be A_r. After that, there are two cases:

- If $A_{r+1} > A_r$ (or $r = N$), we can choose any $r - 1$ positions out of the remaining $N - 1$ and place the elements $A_1, A_2, \dots, A_{r-1}$ in these positions (in descending order) and the other elements in the remaining positions (in ascending order). So, there are $\binom{N-1}{r-1}$ ways for this to happen.
- If $A_{r+1} = A_r$, we need to be a bit more careful. We cannot arbitrarily choose $r - 1$ positions to place the first $r - 1$ elements, because we need to ensure that A_{r+1} is *strictly* the maximum when we place it. This means we need to place at least one element less than A_{r+1} before placing it.
  So, let $x < r$ be the largest index such that $A_x < A_r$. We first place the elements $A_r, A_{r-1}, \dots, A_x$ in the first $r - x + 1$ positions, then we can choose $x - 1$ positions from the remaining $N - (r - x + 1)$ to place the other elements, giving us $\binom{N-r+x-1}{x-1}$ possible ways.
  x can be maintained as you iterate through the array: update it whenever you move from one value to the next.

Adding this up across all positions gives the final answer.

Each binomial coefficient needs to be computed in $\mathrm{O}(\log MOD)$ or $\mathrm{O}(1)$. If you don't know how to do this, read through **this article**.

TIME COMPLEXITY:

$\mathrm{O}(N \log N)$ per testcase.

PREREQUISITES:

Sieve of Eratosthenes

PROBLEM:

You're given an array of N integers, each between 1 and 10^6.
Answer Q queries of the following form on this array:

- Given an integer x, find the smallest integer y present in A such that $\gcd(x, y) > 1$, print it, and delete it from A.
- If no such y exists, then print and delete the smallest element of A instead.

EXPLANATION:

The main observation required to solve this problem is the fact that, if $\gcd(x, y) > 1$, then there must exist a *prime number* p such that p divides both x and y.

So, to answer a given query x, we can do the following:

- Quickly all prime factors of x, say they're $p_1, p_2, \ldots, p_k$.
- For each p_i, find the smallest element of the array that's divisible by p_i, say y_i.
- The answer is then simply $\min(y_1, y_2, \ldots, y_k)$.
- Of course, if none of the y_i exist simply take the smallest element of the array.

The main hurdle is now how we would do this quickly. Prime factorization in $\mathrm{O}(\sqrt{x})$ each time is almost certainly too slow, since we need to factorize several numbers (all the array elements and the queries).
Notice the constraint $A_i \le 10^6$, which we'll use to our advantage.

Prime factorizing small numbers fast

Notice that for this problem, we don't really need all the factors of a number: we only need its *prime* factors.
In particular, a number x can only have $\le \log x$ distinct prime factors, since $2^{\log x} = x$.
This gives us hope that maybe we can find them all quickly.

Since $A_i \le 10^6$, let's do some preprocessing.
Using a (slightly modified) sieve of Eratosthenes, one can find, for each $1 \le x \le 10^6$, a list of all its prime factors.
This can be done in $\mathrm{O}(M \log \log M)$, where $M = 10^6$ here.

Once these lists are stored, prime factorizing a number is simple: just iterate through the list!

Now that we're done with the prime factorization, let's move on to answering queries.
We need to support the following:

- For a given prime p, find the smallest element of A that's divisible by p.
- Delete an element from A, which also requires us to update the information of all primes it's divisible by.

So, we need a data structure that supports quick insertion/deletion, and quickly finding the minimum.
The easiest way to achieve this is by using `std::set`/`std::multiset` in C++ or `TreeSet` in Java. Coding in Python will require you to be a bit more clever, though it's still doable — see the editorialist's code linked below.

At any rate, pick your data structure of choice, and assign one to each prime, using say a map.
Then,

- For each i, quickly prime factorize A_i using the approach discussed above. For each prime p that divides A_i, insert A_i into its corresponding set.

- Notice that this uses $\mathrm{O}(N \log 10^6)$ memory, since each A_i has $\leq \log A_i$ prime factors and so will only be present in that many lists.
- Then, for each query x,
  - Prime factorize x as discussed.
  - For each prime factor, find the smallest remaining element; after which you'll know which value needs to be deleted from the array
- To delete an element, simply reverse the insertion process: find each prime factor, and delete A_i from the appropriate list.

Each element is inserted into at most $\log A_i$ lists and deleted at most once from each, so the complexity of this part is $\mathrm{O}(N \log 10^6 \log N)$; perhaps with an extra $\log$ depending on implementation.
Each query requires us to iterate through $\leq \log x$ lists; and query each one for the minimum which can be done in $\mathrm{O}(1)$. So, the complexity of this part is $\mathrm{O}(Q \log 10^6)$, again with maybe an extra $\log$ from a map.

TIME COMPLEXITY

$\mathrm{O}(M \log \log M)$ precomputation, followed by $\mathrm{O}((N+Q) \log N \log M)$ per test case, where $M = 10^6$.

PREREQUISITES:

Tries, basic probability, DFS

PROBLEM:

There is a tree on N vertices, the i-th vertex has value A_i. Alice starts at vertex 1 and randomly moves to a child of the current vertex till she can no longer do so.

Her score is the bitwise xor of all but one of the values she visited along the way, and she can choose which one to drop in order to maximize her score.
What's her expected final score?

EXPLANATION:

Applying the definition of expected value, we see that the answer is $\sum P_u S_u$, where:

- P_u is the probability that Alice reaches u
- S_u is the score obtained by Alice if she reaches u

and the above summation is taken across all terminal vertices (i.e, leaves) in the tree.
Let's compute them individually.

Computing probabilities

Let P_u be the probability that Alice visits vertex u during her journey.
Since she starts at 1, we have $P_1 = 1$.

Now, consider some $u > 1$ that has c children. Let v be one such child.
If we already know P_u, computing P_v is easy: it's just $\frac{P_u}{c}$. This is because Alice must reach u then choose to go into v, so we multiply the probabilities together.

This gives us a way to compute P_u for every vertex u using DFS, starting from the root to the leaves; and the whole process takes $\mathrm{O}(N \log 10^9)$ time (the log factor because we perform modulo divisions).

Computing score

Let u be a leaf vertex. Let's compute Alice's score when she reaches it.

Suppose the values on the path to u are $x_1, \ldots, x_k$. We want the maximum value of

$$x_1 \oplus \ldots \oplus x_{i-1} \oplus x_{i+1} \oplus \ldots \oplus x_k$$

across all i.
Notice that this is just

$$(x_1 \oplus \ldots \oplus x_{i-1} \oplus x_i \oplus x_{i+1} \oplus \ldots \oplus x_k) \oplus x_i$$

and the first value is a constant, being the xor-sum of all the values.

So, we have a list $[x_1, \ldots, x_k]$ and a constant C, and we'd like to find the maximum possible value of $C \oplus x_i$ for some i.

This is a well-known problem, and can be solved using a trie: a tutorial can be found **here**.

However, the values in the trie keep changing, and we can't afford to insert the entire path each time we process a leaf.

Instead, we maintain a single trie and reuse it across our DFS, as follows:

- Let T denote our trie. Initially, it's empty.
- When we enter node u, insert A_u into T.
- Then, if u is a leaf perform the relevant query; and if not continue the DFS into the children of u
- Finally, when exiting u, remove (one copy of) A_u from T.

This ensures we perform only N insertions and deletions each across the whole process, making the time complexity $\mathrm{O}(N \log 10^9)$.

Once both the probabilities and the scores have been computed, calculating the final answer is trivial using the summation above.

TIME COMPLEXITY:

$\mathrm{O}(N \log 10^9)$ per testcase.

PREREQUISITES:

Observation, dynamic programming

PROBLEM:

You're given N intervals $[L_i, R_i]$ where $L_i < R_i$.

A *good operation* is as follows:
Pick M of these intervals, and $K_i > 1$ distinct integers $Y_{i,1}, Y_{i,2}, \ldots, Y_{i,K_i}$ from the i-th chosen interval such that:

- The product $K_1 \times K_2 \times \ldots \times K_M$ doesn't exceed N.
- M is as large as possible.

The score of a good operation is the product of all chosen $Y_{i,j}$, i.e,

$$\prod_{i=1}^{M} \prod_{j=1}^{K_i} Y_{i,j}$$

Find the sum of scores across all possible good operations.

EXPLANATION:

There are a lot of parts to this problem, so let's analyze them one at a time.

Good operations

First, let's look at when exactly an operation is good.
We pick M intervals and at least 2 elements from each chosen interval.
The product of the numbers of chosen elements shouldn't exceed N, which already tells us that M can't be very large since this product is at least 2^M.
In particular, this gives us the upper bound $M \leq \lfloor log_2 N \rfloor$.

Since M should be maximum, we can always choose $M = \lfloor log_2 N \rfloor$ (we can just pick 2 elements from each interval).
So, M is fixed.

Now, let's look at the K_i values.
Clearly most of them should be 2, but if $2^M < N$ we have *some* leeway.
For example, if $N = 6$ we have $M = 2$, and we have $2 \cdot 2 \leq 6$ and $2 \cdot 3 \leq 6$, which means we can have one of the K_i equal to 3 if we like.

However, despite this leeway, the restrictions are in fact fairly tight.
Specifically, a little analysis should tell you the following:

- Every K_i will be ≤ 3.
- At most one K_i can be 3.

Proof

Suppose $K_i \geq 4$.
Then, we can instead pick 2 elements each from 2 other intervals for a larger M but not larger product, which is surely not optimal.

Similarly, if we have 2 intervals with $K_i = 3$, their product is 9 and we can instead pick 3 intervals and choose 2 from each for larger M and lower product, which once again isn't optimal.

This puts a rather strict structure on what a good operation looks like:

- We always choose exactly $M = \lfloor log_2 N \rfloor$ intervals.
- Then, either we pick two elements from all M of these, or (product allowing) we pick three elements from one of them and two from all the rest.

This allows us to compute the score using dynamic programming.

Computing the sum of scores

There are two cases here: either we pick 2 elements from every interval, or we pick two elements from all but one and 3 from the last one.
Dealing with both is similar, so let's see how to answer the first case (i.e, all twos) first.

To get an idea of how to do do this, let's look at a simple case first: there are two intervals $[L_1, R_1]$ and $[L_2, R_2]$, and we pick exactly two elements from each one.
How would we compute the sum of scores in this case?

Well, suppose we pick the elements x_1, x_2 from the first interval and x_3, x_4 from the second. They add $x_1 x_2 x_3 x_4$ to the sum.
Summing this across all possible values, we want

$$\begin{aligned}
&\sum_{L_1 \le x_1 < x_2 \le R_1} \sum_{L_2 \le x_3 < x_4 \le R_2} (x_1 x_2 x_3 x_4) \\
&= \sum_{(x_1, x_2)} \left(x_1 x_2 \sum_{(x_3, x_4)} x_3 x_4 \right) \\
&= \left(\sum_{(x_1, x_2)} x_1 x_2 \right) \left(\sum_{(x_3, x_4)} x_3 x_4 \right)
\end{aligned}$$

That is, if we know the sum of pairwise products of each the two intervals, all we'd need to do is multiply them.

It's easy to see that this generalizes to M intervals as well: if P_i is the sum of pairwise products of the i-th interval, then what we're looking for is the sum of products of the P_i values taken M at a time.

First, let's compute the P_i values. That can be done with a bit of math.
For an interval $[L_i, R_i]$, the corresponding P_i value is

$$\frac{(L_i + L_{i+1} + \ldots + R_i)^2 - (L_i^2 + L_{i+1}^2 + \ldots + R_i^2)}{2}$$

because:

- $(L_i + L_{i+1} + \ldots + R_i)^2$ is the product of all pairs of values in the interval, disregarding order or distinctness.
- We subtract $(L_i^2 + L_{i+1}^2 + \ldots + R_i^2)$ to account for choosing two of the same element.
- Now the remaining value didn't account for order, i.e, $x_1 x_2$ and $x_2 x_1$ were both counted as different products.
  To account for this, divide by 2.

Once all the P_i values are computed, we want to compute their product taken M at a time.
That can be done in $\mathrm{O}(N \cdot M)$ with a fairly simple DP.

Let $\mathrm{dp}[i][j]$ denote the sum of products all j-tuples of the first i values. Then,
$\mathrm{dp}[i][j] = \mathrm{dp}[i-1][j] + P_i \cdot \mathrm{dp}[i-1][j-1]$
The value we're looking for is, of course, $\mathrm{dp}[N][M]$.

As a small side note, the sum of products of the elements taken M at a time can be computed for *every* M from 1 to N in $\mathrm{O}(N \log^2 N)$ using NTT, but that's extreme overkill for this problem because we're only interested in a single value of M, and the $\mathrm{O}(N \cdot M)$ method is enough because $M = \lfloor \log_2 N \rfloor$.

This almost solves the problem: the only remaining case is when we take exactly three elements from one interval.
You might've noticed that this is not so different from the above case: indeed, once again we only really need to know, for each interval, the sum of products of all distinct triples of elements in it.

This can be computed almost exactly the same way we computed P_i, by using inclusion-exclusion.

Details

Just as before, we start with $(L_i + \ldots + R_i)^3$, disregarding both distinctness and order.

From this, subtract $3 \cdot (L_i + \ldots + R_i) \cdot (L_i^2 + \ldots + R_i^2)$, which is the sum of all triplets such that two elements are equal (fix the equal pair in $\binom{3}{2} = 3$ ways, then it's the sum of all elements multiplied by the sum of squares of all elements).

Next, add $2 \cdot (L_i^3 + \ldots + R_i^3)$, to account for the case when all three elements are equal.
Such cases were subtracted thrice above but should've been subtracted only once; hence the multiplier of 2.

Now we're left with the sum of products of triplets, all of whose elements are distinct.
To account for order, divide this by 6 to get the final answer.

Note that in intermediate steps, we needed to compute the sum of first/second/third powers of a range of integers; there are well-known formulas for these that allow for $\mathrm{O}(1)$ computation.

Once these values are computed for each interval (let's call them Q_i), the earlier dp can be easily modified to account for them.
Let $\mathrm{dp}[i][j][0/1]$ denote the sum of scores for the first i intervals if we've chosen exactly j so far, and the third dimension denotes whether we have/haven't chosen 3 from an interval yet.
Then,

- $\mathrm{dp}[i][j][0] = \mathrm{dp}[i-1][j][0] + \mathrm{dp}[i-1][j-1][0] \cdot P_i$
- $\mathrm{dp}[i][j][1] = \mathrm{dp}[i-1][j][1] + \mathrm{dp}[i-1][j-1][1] \cdot P_i + \mathrm{dp}[i-1][j-1][0] \cdot Q_i$

Once again, this takes $\mathrm{O}(N \log N)$ time.
It can also be implemented with $\mathrm{O}(\log N)$ memory, though that's likely unnecessary to get AC.

As for the final answer,

- If $3 \cdot 2^{M-1} \le N$, we can have an interval with $K_i = 3$. In this case, the answer is $\mathrm{dp}[i][M][0] + \mathrm{dp}[i][M][1]$.
- Otherwise, all intervals must have $K_i = 2$ and the answer is just $\mathrm{dp}[i][M][0]$.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PROBLEM:

You have two arrays A and B.
Find the number of permutations P such that the following holds:

- For each i, you can choose either the value $A_i + B_{P_i}$ or $A_i - B_{P_i}$
- There must exist a way to make these choices such that all N values obtained are equal.

EXPLANATION:

First, let's replace each B_i by $|B_i|$.
This makes reasoning about things easier, and doesn't actually affect what we can do since at worst signs are flipped, but we have the choice of both addition and subtraction to counteract that.

For convenience, let's also sort the arrays A and B, so that $A_i \le A_{i+1}$ and $B_i \le B_{i+1}$. Recall that every B_i is currently ≥ 0 as well.

Now, B_N has to be matched with something, say A_i.
There are two cases:

- Suppose we choose $A_i + B_N$. Then, if there exists a j such that $A_j < A_i$, it's impossible to make all the elements equal (since $A_j + B_k$ can never equal $A_i + B_N$ no matter what k is chosen).
  This means that in this case, A_i must be the **minimum** element of the array.
- Suppose we choose $A_i - B_N$. Similar reasoning tells us that A_i should be the maximum element of the array in this case.

So, there are only two cases to check: making everything equal to $A_1 + B_N$, or making everything equal to $A_N - B_N$.
Note that if $A_1 + B_N = A_N - B_N$, only one check needs to be made to prevent double-counting.

We now have a subproblem to solve: suppose the final value should be x. How many permutations allow us to achieve this?

This can be solved as follows:

- Let freq denote the frequency array of B, so $\text{freq}[x]$ denotes the number of times x occurs in B.
- Initialize the answer to 1.
- For each i from 1 to N,
  - If $A_i < x$, we need to match it with $x - A_i$. Otherwise, we need to match it with $A_i - x$.
    Note that this is because we made all the $B_i \ge 0$, which makes this step simpler since we only need to care about positive numbers.
  - In general, we need to match A_i with $y = |A_i - x|$.
  - There are $\text{freq}[y]$ such choices, so multiply the answer with $\text{freq}[y]$.
  - Then, decrease $\text{freq}[y]$ by 1, to account for the one we used up.

So, for a fixed x, we can compute the number of valid permutations in $\text{O}(N)$ or $\text{O}(N \log N)$.
There are at most 2 values of x that need to be checked for, so check both and add up the answers to obtain the final answer.

TIME COMPLEXITY

$\text{O}(N)$ or $\text{O}(N \log N)$ per test case

PROBLEM:

You're given an $N \times M$ grid A.
You can choose a cell (i,j), delete the row i and column j, and compute the sum of $A_{i,j} \oplus x$ across x remaining in the grid.

Find the maximum possible value of this sum if (i,j) is chosen optimally.

EXPLANATION:

Let's fix a cell (i,j) and try to quickly compute the resulting sum.

Iterating over all remaining elements would be too slow, we need to do a bit better.
As always, when dealing with bitwise operations, it helps to look at each bit individually.

So, let's fix a bit b, and see how many of the $A_{i,j} \oplus x$ values have this bit set.
If this count is k_b, then bit b contributes $k_b \cdot 2^b$ to the overall sum.
We only care about values of b upto 30, so if we can find k_b for each one quickly we'll be done.

There are two cases:

- **Case 1:** $A_{i,j}$ itself has bit b set.
  Here, $A_{i,j} \oplus x$ will have b set if and only if x *doesn't* have b set.
  So, we'd like to know the total number of x that don't have bit b set.
- **Case 2:** $A_{i,j}$ doesn't have bit b set.
  Here, $A_{i,j} \oplus x$ will have b set if and only if x has b set.
  So, we'd like to know the total number of x that do have bit b set.

Notice that both cases require us to know similar information: for fixed (i,j) and bit b, how many elements outside the i-th row and j-column have (or don't have) bit b set.

This can be found in $\mathrm{O}(1)$ with a bit of precomputation.
In particular,

- Let ct_b be the total number of values in the grid that have bit b set.
- Let $\mathrm{row}_{i,b}$ the the number of values in the i-th row that have bit b set.
- Let $\mathrm{col}_{j,b}$ the the number of values in the j-th column that have bit b set.
- Then, the number of elements outside row i and column j that have bit b set equals
  $ct_b - \mathrm{row}_{i,b} - \mathrm{col}_{j,b} + (1 \text{ if } A_{i,j} \text{ has bit } b \text{ set, and } 0 \text{ otherwise})$.
- The number of elements that *don't* have bit b set equals $(N-1) \cdot (M-1)$ minus the above quantity.

The values of ct_b, $\mathrm{row}_{i,b}$ and $\mathrm{col}_{j,b}$ can be precomputed for all i, j, b by going through the grid once.

After this, the answer for each (i,j) can be found by iterating across each bit b and computing k_b.
We check 30 bits for each cell, so the overall complexity is $\mathrm{O}(30 \cdot NM)$, which is fast enough.

TIME COMPLEXITY

$\mathrm{O}(30 \cdot NM)$ per testcase.

PREREQUISITES:

Sweep line, ternary search and fenwick/segment tree *or* two pointers

PROBLEM:

N people are on a grid, the i-th at position (x_i, y_i).

Find the smallest total cost required to make K of them meet at a single point; where the distance is Manhattan distance.

EXPLANATION:

First off, it should be obvious that the optimal meeting point, say (a, b), will be such that a is the x-coordinate of one of the people and b is the y-coordinate of one of the people.

This gives us N^2 potential meeting points, so we need to be able to compute the best answer for them all.

For a fixed meeting point (a, b), computing the answer in $\mathrm{O}(N)$ or $\mathrm{O}(N \log N)$ isn't too hard: compute the distance of all N points from this one, and take the K smallest of them.
That's obviously too slow though.

Let's fix the value of b and sweep the value of a from small to large.
This has the following effect as a varies:

- The distance of (x_i, y_i) from (a, b) is $|x_i - a| + |y_i - b|$.
- In particular, this is $x_i - a + |y_i - b|$ if $x_i \geq a$ and $a - x_i + |y_i - b|$ otherwise.
  Note that the second part of the sum ($|y_i - b|$) remains constant as a changes.

So, once b is fixed, each point has a 'base' value $|y_i - b|$, and a 'dynamic' value, which is either $x_i - a$ or $a - x_i$ depending on the current value of a.

Let's say a point is on the *left* if $x_i < a$, and on the *right* otherwise.
If we sweep a from low to high, every point initially starts out on the right, and then one by one points move to the left.

Let L denote the multiset of values (i.e, base value + dynamic value) of the left points, and R denote the same for the set of right points.
Once L and R are fixed (which is unique when a and b are fixed), our aim is then to find k_1 and k_2 such that:

- $k_1 + k_2 = K$
- $S_1 + S_2$ is minimized, where
  - S_1 is the sum of the k_1 smallest values in L
  - S_2 is the sum of the k_2 smallest values in R

We have two objectives: maintain L and R appropriately, and find the optimal k_1 and k_2.
There are a couple of ways to do this.

Setter's method (two pointers)

Suppose we know the k_1 and k_2 values for a fixed a and b.
Let's move the sweep line by one step, say to a'.

This moves a few points from R into L, so let's do that.
Suppose c points were moved.

The main observation here is that after this movement, the new optimal k_1 value won't change too much: in fact, it'll change by $\mathrm{O}(c)$.

This allows us to 'adjust' the k_1 and k_2 values by maintaining pointers to their current values, and moving them by $\mathrm{O}(c)$ to compute the new optimum.
This way, L and R can be maintained as multisets while k_1 and k_2 can be kept as iterators to these multisets; so that they remain sorted after insertion/deletion.

The implementation of this can be found in the setter's code below.
Editorialist's method (ternary search + fenwick)

Let $f(k_1)$ denote the cost of taking the smallest k_1 elements from L and smallest k_2 elements from R.

Intuitively, as we move k_1 from 0 to K, $f(k_1)$ should decrease, hit a minimum, then increase again.
This allows us to ternary search on f to find the minimum $f(k_1)$.

In particular, evaluating $f(k_1)$ requires us to quickly compute the sum of the smallest k_1 elements in L and smallest k_2 elements in R.

So, we need to maintain L and R in a data structure that supports:

- Elements can be quickly added/removed from the set
- Query for the sum of the smallest k_1 elements of the set

This can be accomplished with a Fenwick tree, built on values.

First, precompute all the 'left' values for all the points, and put them in sorted order.
Build a Fenwick tree on this sorted order, with each element initially zero.
Do the same thing for the 'right' values, but this time populate the fenwick tree with the appropriate values as well.

Now,

- Removing an element from the 'right' Fenwick tree just means setting the corresponding position to 0
- Adding an element to the 'left' Fenwick tree means setting the value to the corresponding position
- Computing the sum of the smallest k_1 non-zero values in a Fenwick tree is also doable in $\mathrm{O}(\log N)$ with binary search.

This gives us a solution in $\mathrm{O}(N^2 \log^2 N)$, since each position requires $\mathrm{O}(\log^2 N)$ work and there are N^2 positions to consider.
Moving from one position to the next during the sweep is $\mathrm{O}(\log N)$ work which is dwarfed by the ternary search.

TIME COMPLEXITY

$\mathrm{O}(N^2 \log N)$ or $\mathrm{O}(N^2 \log^2 N)$ per test case.

PREREQUISITES:

Prefix sums, dynamic programming

PROBLEM:

Given a binary array A, find the longest *good* sequence $x_1, x_2, \ldots, x_k$ such that it satisfies:

- $1 \le x_1 < x_2 < \ldots < x_k \le N + 1$
- $A[x_i : x_j - 1]$ contains an equal number of zeros and ones for each $i < j$.

EXPLANATION:

First, note that it's enough to ensure that $A[x_i : x_{i+1} - 1]$ (i.e, the subarray between consecutive elements) satisfies the condition, i.e, contains an extra one.

Proof

This is not hard to see.
The subarray $A[x_i : x_j - 1]$ can be thought of as the concatenation of $A[x_i : x_{i+1} - 1], A[x_{i+1} : x_{i+2} - 1], \ldots, A[x_{j-1}, x_j - 1]$. Note that there are $j - i$ such subarrays.

If each of these contains an extra one, of course all together the number of extra ones will be exactly $j - i$.

Now, let's use a small trick: replace every 0 in the array with -1. Let's call the array obtained this way as B.
For example, if $A = [0, 1, 1, 0, 1]$ then $B = [-1, 1, 1, -1, 1]$.
Notice that "$A[L : R]$ contains an extra one" translates to "$B[L : R]$ has a sum of 1" in the new array, which is much easier to deal with!

Since we're dealing with subarray sums, it's natural to think of prefix sums.
So, let $P_i = B_1 + B_2 + \ldots + B_i$ (with $P_0 = 0$) denote the prefix sum array of B.

Now, for $B[L : R]$ to have a sum of 1, we must have $P_R - P_{L-1} = 1$, or $P_R = P_{L-1} + 1$.

Applying this to the definition of a good sequence, we see that we must have $P_{x_{i+1}-1} = P_{x_i - 1} + 1$ for every $1 \le i < k$.
In other words, we need to choose a sequence of indices whose prefix sums are $k, k+1, k+2, \ldots$ for some value of k.

The longest good sequence thus corresponds to the longest sequence of prefix sums of this form.

This can be computed using dynamic programming.

Let dp_i denote the length of the longest sequence of prefix sums ending at position i that satisfies the above condition.
Transitions for a given index are quite easy to compute in $\mathrm{O}(N)$. We simply have $dp_i = 1 + \max(dp_j)$ across all $j < i$ such that $P_j + 1 = P_i$.

However, doing this $\mathrm{O}(N)$ computation for each index is too slow, we need to speed it up a bit.
This can be done by noting that we only care about those indices whose P_j values equal $P_i - 1$; in fact, we only care about the maximum dp_j value among these indices.

So, let's keep another array mx, where $\mathrm{mx}[x]$ is the largest value of dp_i across all indices i such that $P_i = x$.
Initially, initialize it to all zeros.
Then, for each i from 0 to N,

- $dp_i = 1 + \mathrm{mx}[P_i]$

- $\text{mx}[P_i] = \max(dp_i, \text{mx}[P_i])$

The maximum element of dp is the final answer.
Reconstructing a valid sequence is not hard, when performing the dp keep a link to the previous element in the sequence, then follow these in reverse at the end.

TIME COMPLEXITY:

$\text{O}(N)$ per testcase.

PROBLEM:

Alice and Bob play a game.
Carol gives Alice an integer A and Bob an integer B, such that $1 \le A, B \le N$ and $\max(A, B)$ is a strictly larger multiple of $\min(A, B)$.

Alice and Bob take turns playing, Alice goes first.
On their turn, a player does the following:

- If they certainly know the value of $\min(A, B)$, they'll say it and the game ends.
- Otherwise, they say "I don't know" and the turn passes to the other player.

If any valid (A, B) pair can be chosen equally randomly, find the expected number of turns the game takes.
Note that Alice and Bob only know their numbers (and subsequently, whatever the other person says) — in particular, neither player knows N.

EXPLANATION:

First, let's analyze who wins a single instance of the game and how many turns it takes.

- If $A = 1$, Alice knows that $\min(A, B) = 1$ for sure and wins.
  This took 1 turn.
- Otherwise, $A > 1$, but Alice has no further information: maybe $B = 1$, and she wouldn't know.
  So, the turn passes to Bob.
- On Bob's turn, he knows that $A \neq 1$ for sure.
  If $B = 1$, he knows $\min(A, B) = 1$ and the game ends in 2 turns.
  Further, if B is a prime, Bob still knows that $\min(A, B) = B$. This is because Alice doesn't have 1, and so whatever she has must be a multiple of the prime B.
- If B is neither 1 nor a prime, Bob cannot answer definitively - there's no way of him knowing if *Alice* has a prime, for example. So, the turn passes to Alice.
- Now, Alice knows that Bob has neither 1 nor a prime.
  So, if she has a prime herself, $\min(A, B)$ is known.
  Further, if A has two prime factors (i.e $A = pq$ or $A = p^2$ for primes p, q) then Alice knows that $\min(A, B) = A$; because Bob's number will also have at least two prime factors, and if A has two, then B *must* have more than 2.
- If not, the turn passes to Bob, and so on.

It can be observed that the number of turns depends on the number of prime factors of $\min(A, B)$.
In particular, if players have said "I don't know" x times, that eliminates all numbers with $< x$ prime factors; so if your own number has x prime factors you know it's the minimum.

So, if $\min(A, B)$ has k prime factors,

- Suppose $A = \min(A, B)$. Then,
  - if k is even, Alice wins on the $(k + 1)$-th move.
  - Otherwise, Alice wins on the $(k + 2)$-th move (for example, $A = 4$ and $B = 8$).
- Suppose $B = \min(A, B)$. Then,
  - If k is odd. Then, Bob wins on the $(k + 1)$-th move.
  - Otherwise, Bob wins on the $(k + 2)$-th move.

Notice that for any valid pair (x, y), the pairs (x, y) and (y, x) together contribute $2k + 3$ moves, where k is the number of prime factors of $\min(x, y)$.

This already gives us a solution in $\mathrm{O}(N \log N)$.

- iterate across all pairs of integers $(a, d \cdot a)$ where $1 \le a < d \cdot a \le N$
  There are $\mathrm{O}(N \log N)$ such pairs.
- For each such pair, let k be the number of prime factors of a.
  k can be computed for all integers $\le N$ with a sieve.
  Then, the pairs $(a, d \cdot a)$ and $(d \cdot a, a)$ contribute $2k + 3$ moves in total.
- This tells us both the total number of moves and the number of valid pairs, so their ratio is the answer.

This will pass subtask 1.

Subtask 2 follows the same general idea as subtask 1, but some optimizations are needed.

First, the sum of N across all testcases is no longer bounded.
We'll precompute the answers for all $1 \le N \le 10^7$ and just output the answer when asked.

However, the precomputation itself can't be $\mathrm{O}(N \log N)$ time: that will likely be too slow.

We need to find two things:

- First, the contribution of each pair $(a, d \cdot a)$, to form the numerator.
- Second, the number of proper divisors of each number, to form the denominator.
  A proper divisor of a number is a factor that's strictly less than it.

The total number of divisors can be found faster than $\mathrm{O}(N \log N)$ by counting it from the prime factorization of each number.
That is, if $N = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$, it has $(a_1 + 1) \cdot (a_2 + 1) \cdot \dots \cdot (a_k + 1)$ divisors in total; from which we subtract 1 to obtain the count of proper factors.

The prime factorization can be found in $\mathrm{O}(N \log \log N)$ time by sieving only across primes and storing a prime divisor of each number.
This allows us to find divisor counts in $\mathrm{O}(N \log \log N)$ time too.

Now, let's look at the numerator.
Let $\mathrm{pct}[a]$ denote the number of prime factors of a (which we already computed, above).
Instead of fixing $(a, d \cdot a)$ and adding $2\mathrm{pct}[a] + 3$ to the total, let's fix the larger number and count the contribution of all its factors to it.

Consider the integer n.
For each proper divisor d of n, we add $2\mathrm{pct}[d] + 3$ to the answer of all integers $\ge n$.

Let's take the 3 out: if n has k proper divisors, it'll contribute a total of $3k$.
So, we only really need to know the sum of $\mathrm{pct}[d]$ across all proper factors of n.

Equivalently, we can find for each prime factor in the factorization of n, the number of factors it appears in.
Summed up across all d, it can be seen that this value is nothing but $\frac{1}{2} \cdot \mathrm{pct}[n] \cdot \mathrm{divs}[n]$.
This includes case when n is treated as a divisor of itself, so make sure to subtract that later on.

Notice that we've already computed pct and divs, which means the above formula can be computed in $\mathrm{O}(1)$ time for a fixed n.
Let $\mathrm{val}[n]$ denote the required value for a fixed n (including the $3 \cdot (\mathrm{divs}[n] - 1)$ we took out earlier.

Then, for a given N,

- The numerator is $\mathrm{vals}[1] + \mathrm{vals}[2] + \dots + \mathrm{vals}[N]$
- The denominator is $\mathrm{divs}[1] + \mathrm{divs}[2] + \dots + \mathrm{divs}[N]$

Both of these can be precomputed using prefix sums, and their ratio gives the answer!

TIME COMPLEXITY

$O(N \log \log N + T \log MOD)$, where $N = 10^7$.

PROBLEM:

There is a function F such that $F(i)$ is predetermined (but secret) for $0 \le i < K$, and $F(i) = -\sum_{j=1}^{K} F(i-j)$ otherwise.

In one query, you can provide x and the judge will return $F(x)$.

You are given an array B of length N. Using the least possible number of queries, compute the values of $F(B_1), F(B_2), \ldots, F(B_N)$.

EXPLANATION:

Playing around with the function a little should give you a rather useful piece of information: F is periodic, with period $K+1$. That is, $F(i) = F(i+K+1)$ for every $i \ge 0$.

Proof

Let's try computing some values of F.

$F(0), F(1), \ldots, F(K-1)$ are all fixed.
The formula gives us $F(K) = -(F(0) + F(1) + \ldots + F(K-1))$.
Now try computing the next few values:

- $F(K+1) = -(F(1) + F(2) + \ldots + F(K)) = F(0)$ if you substitute the value of $F(K)$ from above.
- $F(K+2) = F(1)$ using a similar computatation

  $\vdots$
- $F(2K) = F(K-1)$
- $F(2K+1) = -(F(2K) + F(2K-1) + \ldots + F(K+1)) = -(F(0) + F(1) + \ldots + F(K-1))$

Notice that these $K+1$ values are exactly equal to the first $K+1$.
Now that the values have repeated, apply the same argument to this batch of $K+1$ values to obtain the next $K+1$, and so on.

This tells us that it is enough to know the values of $F(0), F(1), \ldots, F(K)$. We can use these to answer every query, since $F(x) = F(x \bmod (K+1))$.

Notice that the first K values being hidden means we have no choice but to ask for those. However, there are still a couple of optimizations to be made:

First, note that if we know the value of the function at $0, 1, 2, \ldots, K-1$ then we can compute $f(K)$ without a query. This means we can always use $\le K$ queries.

Second, if some value of $x \bmod (K+1)$ doesn't appear in B, we can simply skip asking for it since we're never going to need it anyway.

This gives us our final solution:

- Let $S = \{B_i \bmod (K+1) \mid 1 \le i \le N\}$ be the set of all relevant remainders.
- If the size of S is $K+1$, query for $0, 1, 2, \ldots, K-1$ and compute $F(K)$ from them.
- Otherwise, query only for $F(x)$ such that $x \in S$.
- Using the results of these queries, compute the answer for each B_i.

TIME COMPLEXITY:

$\mathrm{O}(N \log K + K)$ per testcase.

PROBLEM:

There is a hidden permutation P of length N. Find it using at most $N-1$ queries of the following type:

- Provide a permutation Q of length N to the judge.
- The judge will create the permutation $\overline{P}$, where $\overline{P}_i = P_{Q_i}$
- Then, the judge will return the set of prefix maximum positions of $\overline{P}$.

EXPLANATION:

If we are able to find $N-1$ elements of P, that automatically fixes the last element as well.
So, ideally we'd like to be able to find one new element with each operation we perform.

One simple observation that can be made is that the last prefix maximum will always be the position of N, since it's the unique maximum of the permutation.

This allows us to easily find the position of N in P: use $Q = [1, 2, 3, \ldots, N]$, and the last element of the prefix maximum returned is the position of N.

Now that we know where N is, we can choose exactly where to place it in $\overline{P}$.

In particular, suppose we set $\overline{P}_N = N$ and permute the other elements in some way.
Then, the last element of the prefix maximums will be N, and the element before the last will be the position of $N-1$, since the first $N-1$ elements of $\overline{P}$ essentially form a permutation of length $N-1$ on their own.

It's easy to see that this process can be similarly repeated to find $N-2$, then $N-3$, and so on. That is,

- At the start of the i-th iteration, we know the positions of exactly $i-1$ elements, namely $N, N-1, \ldots, N+2-i$.
- Choose Q such that these $i-1$ elements form the suffix of $\overline{P}$, i.e, $\overline{P} = [\ldots, N-2, N-1, N]$
- The prefix maximum array of $\overline{P}$ will have these $i-1$ positions as its last $i-1$ elements; and then the position of $N+1-i$ in $\overline{P}$ as the next one. With this information and Q, the position of $N+1-i$ in P can be found.

In this way, we find the positions of $N, N-1, N-2, \ldots, 2$ in order; after which the position of 1 is known to us.

TIME COMPLEXITY

$\mathrm{O}(N^2)$ per test case.

PREREQUISITES:

DP on trees

PROBLEM:

Given a tree T on N vertices, find the minimum number of edges that need to be added to it so that the resulting graph contains a Hamiltonian path.

EXPLANATION:

Suppose we add some edges to T so that it has a Hamiltonian path.
Notice that the tree edges used in this Hamiltonian path form a set of vertex-disjoint paths in the original tree.

Let's call a set of vertex-disjoint paths a *path forest*.

Of course, minimizing the number of extra edges we use is the same thing as maximizing the number of tree edges forming the complementary path forest; so it seems reasonable that we compute the maximum size of a path forest that can be obtained from the given tree.

Proof of equivalence

Let's call a graph *good* if it contains a Hamiltonian path.
Our claim is that keeping the largest number of edges in a path forest is equivalent to minimizing the number of edges added to form a good graph.
Alternately, the minimum number of edges that need to be *removed* to obtain a path forest equals the minimum number of edges added to form a good graph; which is what we'll prove.

Let P be one minimum (by size) set of edges that, upon removal, form a path forest.
Let H be one minimum (by size) set of edges that, upon addition to T, make it good.

Note that the path forest can be connected into a single long path (thus making the resulting graph good) using $|P|$ additional edges (since removing $|P|$ edges from a tree will create exactly $|P| + 1$ connected components).
So, $|H| \leq |P|$.

Conversely, suppose we add the edges in H.
Consider one Hamiltonian path obtained by this addition.
Suppose there are k tree edges on this path. Then, $|H| = N - 1 - k$ (because H wouldn't be minimal otherwise).
Deleting the $N - 1 - k$ *tree* edges that are not on this path gives us a path forest, so $|S| \leq N - 1 - k = |H|$.

This shows that $|S| = |H|$, and we're done.

Now, notice that in a path forest, every vertex will have degree ≤ 2. This allows us to compute the largest size of a path forest using dynamic programming.

Root the tree at vertex 1.
Then, let $dp(u, d)$ denote the largest size of a path forest in the subtree of vertex u, such that u has degree $\leq d$ in this forest.
Transitions are not too hard:

- $dp(u, 0)$ is simply the sum of $dp(v, 2)$ across all children v of u.
- $dp(u, 1)$ requires us to connect u to one of its children.
  - Suppose we connect it to child c. Then, the size of the path forest is $1 + dp(c, 1) + \sum_{v \neq c} dp(v, 2)$, which also equals $1 + dp(c, 1) - dp(c, 2) + \sum_v dp(v, 2)$.
  - Maximizing this quantity is the same as maximizing $dp(c, 1) - dp(c, 2)$, so take the maximum value across all c and we're done.
- $dp(u, 2)$ requires us to connect u to two of its children.

  - Once again, it's not hard to see that this requires us to take the two largest values of $dp(c, 1) - dp(c, 2)$ across all c; and add them to $dp(u, 0) + 2$.
- Finally because our states were defined to use degree $\leq d$, set $dp(u, 1) = \max(dp(u, 0), dp(u, 1))$ and $dp(u, 2) = \max(dp(u, 2), dp(u, 1), dp(u, 0))$.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Combinatorics

PROBLEM:

You are given a $N \times N$ chessboard. Calculate two things -

- The minimum number of pawns that need to be placed on the board so that they attack every square on the board except the squares on the first rank.
- The number of ways to arrange these pawns in order to achieve the goal modulo $10^9 + 7$.

QUICK EXPLANATION:

This is a casework problem. The following four cases arise -

$N \bmod 4$	Number of pawns needed	Number of ways of arranging them
0	$(N-1) \times \frac{N}{2}$	1
1	$(N-1) \times \frac{N+1}{2}$	$(\frac{N-1}{4})^{N-1}$
2	$(N-1) \times \frac{N+2}{2}$	$(\frac{N+2}{4})^{2(N-1)}$
3	$(N-1) \times \frac{N+1}{2}$	$(\frac{N+5}{4})^{N-1}$

EXPLANATION:

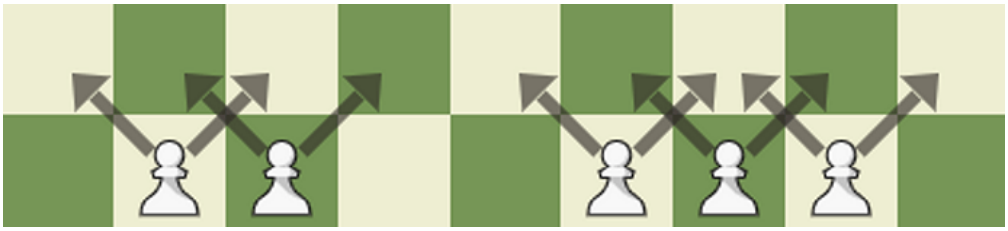
First, notice that a pawn on the ith rank can only attack the squares on the $(i+1)^{th}$ rank. So, the arrangement of pawns in one rank cannot affect the arrangement in any other rank. Hence, the arrangement of pawns in different ranks are independent of each other and we can simply solve the problem for a single rank and then raise it to the power $N-1$ to get the answer for the whole board.

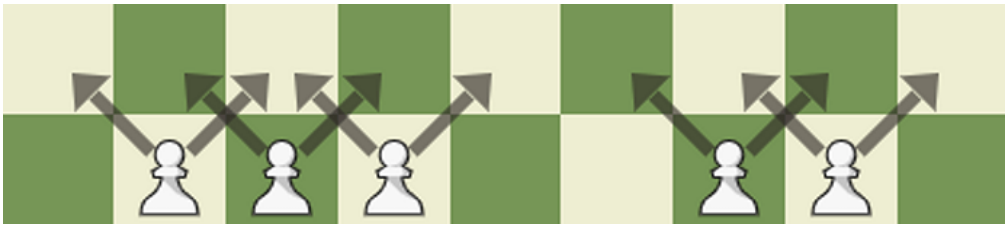
Second, note that a pawn can attack at most two squares having the same color as the square in which the pawn is placed. So, pawns placed on different colored squares are also independent of each other. This leads to the following 4 cases -

1. $N \bmod 4 = 0$: In this case, each rank has $\frac{N}{2}$ light squares and $\frac{N}{2}$ dark squares. So, we need at least $\frac{N}{4}$ pawns to attack the light squares and at least $\frac{N}{4}$ pawns to attack the dark squares. It turns out by construction that these many pawns are sufficient.
   It is also not hard to see that there is only 1 way to arrange them so that they attack all squares on the next rank. So, the second answer is simply 1.
   The following figure shows the arrangement for a single rank of an 8×8 board -

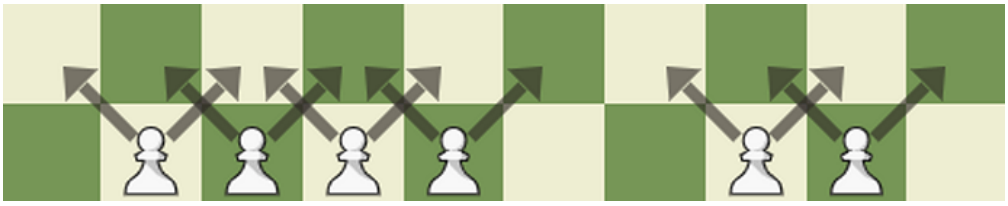
2. $N \bmod 4 = 1$: Without loss of generality, let's assume that there are $\frac{N+1}{2}$ light squares and $\frac{N-1}{2}$ dark squares in the rank we want to attack. Since $N \bmod 4 = 1$, there are an odd number of light squares and an even number of dark squares.
   For the pawns attacking the dark squares, there must be $\frac{N-1}{4}$ such pawns and they can be placed in 1 way only.

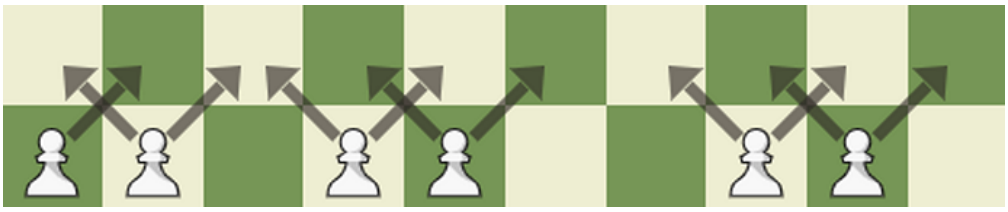
However, for the pawns attacking the light squares, there must be $\frac{N+3}{4}$ such pawns and there are multiple ways to arrange these pawns. It is not hard to see that all pair of consecutive pawns attacking the light squares must be placed at a distance of 4 files from each other except exactly one pair of pawns which must be placed at a distance of 2 from each other. There are $\frac{N+3}{4} - 1 = \frac{N-1}{4}$ choices for the pair.
So, the total number of arrangements for a single rank is $\frac{N-1}{4}$. The total number of arrangements is $(\frac{N-1}{4})^{N-1}$.
The following are the 2 possible arrangements for a single rank of a 9×9 chessboard -

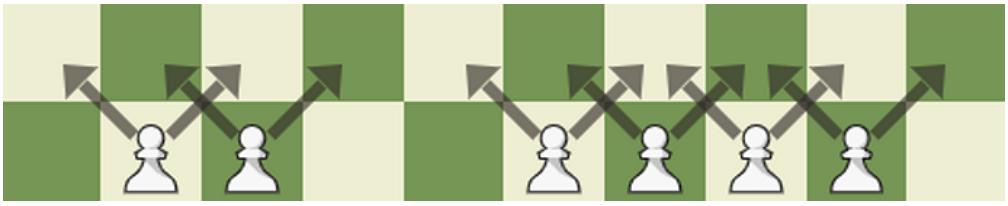


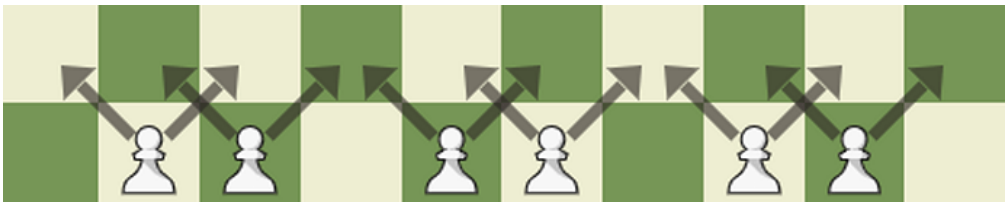


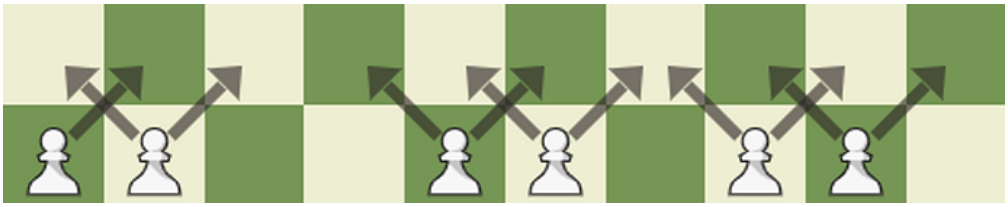
3. $N \bmod 4 = 2$: There are $\frac{N}{2}$ light squares and $\frac{N}{2}$ dark squares in the rank we want to attack. Since $N \bmod 4 = 2$, there are an odd number of light squares and an odd number of dark squares. We need $\frac{N+2}{4}$ pawns to attack squares of each type.
This time, all the pair of consecutive pawns attacking the light squares can be placed at a distance of 4 from each other except at most one pair of pawns which can be placed at a distance of 2 from each other. Same is true for the set of pawns attacking the dark squares. So, there are $\frac{N+2}{4}$ ways to arrange each of these set of pawns (there is 1 way in which no pair of consecutive pawns is at a distance of 2 from each other and $\frac{N+2}{4} - 1$ ways in which exactly one pair of consecutive pawns is at a distance of 2 from each other).
So, the total number of arrangements for a single rank is $(\frac{N+2}{4})^2$. The total number of arrangements for the board is $(\frac{N+2}{4})^{2(N-1)}$.
The following are the 9 possible arrangements for a single rank of a 10×10 chessboard -

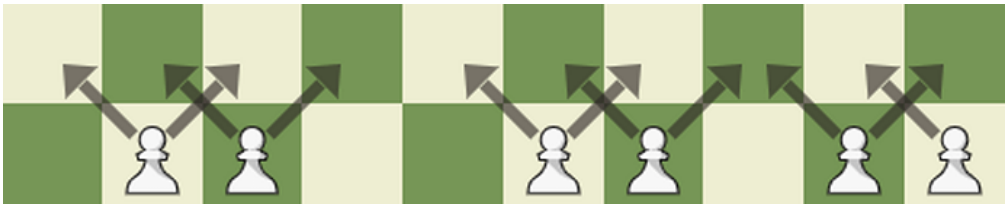


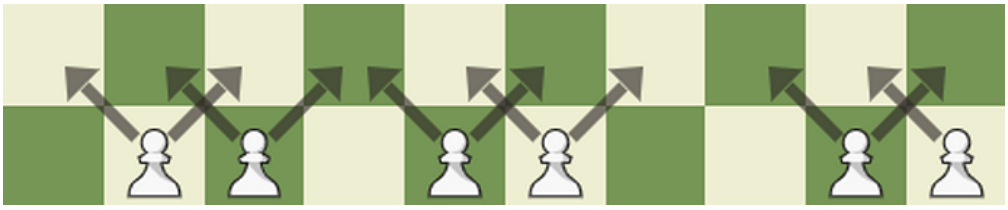


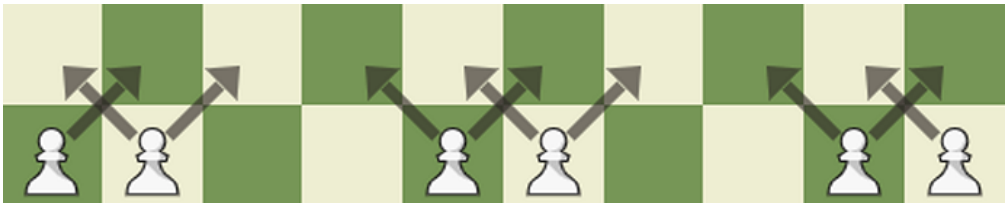




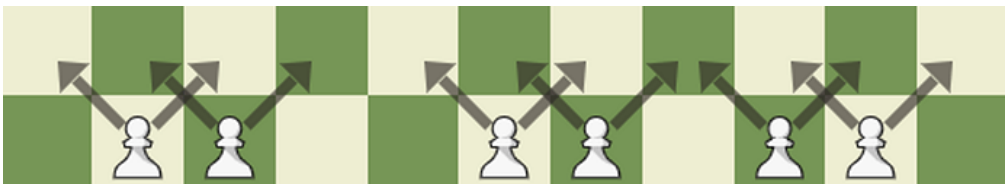


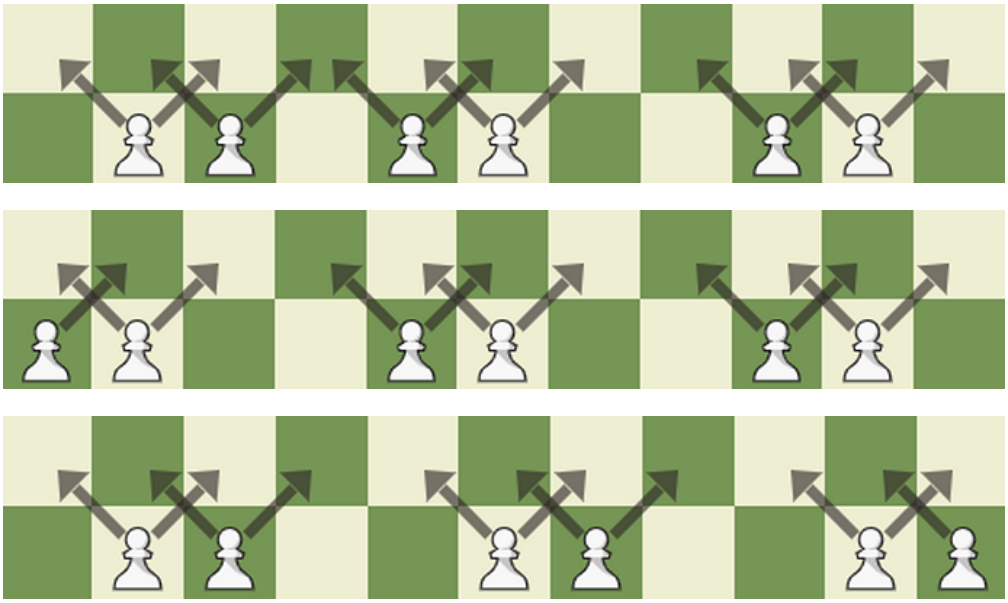






4. $N \bmod 4 = 3$: Without loss of generality, let's assume that there are $\frac{N+1}{2}$ light squares and $\frac{N-1}{2}$ dark squares in the rank we want to attack. Since $N \bmod 4 = 3$, there are an even number of light squares and an odd number of dark squares.
   For the pawns attacking the light squares, there must be $\frac{N+1}{4}$ such pawns and they can be placed in 1 way only.
   However, for the pawns attacking the dark cells, there must be $\frac{N+1}{4}$ such pawns and they can be arranged in $\frac{N+1}{4} + 1 = \frac{N+5}{4}$ ways (there are 2 ways in which no pair of consecutive pawns is at a distance of 2 from each other and $\frac{N+1}{4} - 1$ ways in which exactly one pair of consecutive pawns is at a distance of 2 from each other) .
   So, the total number of arrangements for a single rank is $\frac{N+5}{4}$. The total number of arrangements is $(\frac{N+5}{4})^{N-1}$.
   The following are the 4 possible arrangements for a single rank of a 11×11 chessboard -





TIME COMPLEXITY:

$\mathrm{O}(T \log N)$ because of exponentiation.

PREREQUISITES:

Sorting

PROBLEM:

Given an array A, rearrange it so that every sorted subarray has length ≤ 2.

EXPLANATION:

If $N \leq 2$, any rearrangement works so let's assume $N \geq 3$.

Notice that if $A_i = A_{i+1}$ for some i, then at least one of the subarrays $[A_{i-1}, A_i, A_{i+1}]$ and $[A_i, A_{i+1}, A_{i+2}]$ will exist, and be sorted.

So, if we are to rearrange it, the final array cannot have adjacent equal elements.
Let's use this information to analyze what a valid final array can look like.

Suppose we fix the element A_1. Then, $A_2 \neq A_1$; let's assume $A_2 > A_1$ for now. Note that:

- $[A_1, A_2, A_3]$ shouldn't be sorted, so $A_3 < A_2$ must hold.
- $[A_2, A_3, A_4]$ shouldn't be sorted, so $A_4 > A_3$ must hold.
- $[A_3, A_4, A_5]$ shouldn't be sorted, so $A_5 < A_4$ must hold.

$\vdots$

Notice that this essentially makes A have a 'zig-zag' pattern, i.e,

$$A_1 < A_2 > A_3 < A_4 > A_5 < \dots$$

If $A_2 < A_1$ we get a similar zig-zag pattern, but with the peaks and valleys flipped; either way it's a zig-zag.

Now, let's try to put A into the pattern $A_1 < A_2 > A_3 < A_4 > A_5 < \dots$
Our aim when doing this is to ensure that we never place equal elements next to each other. So, ideally, A_i and A_{i+1} are 'far apart' for every i.

The optimal way to do this is as follows:

- Let S denote the sorted array of A, so $S_1 \leq S_2 \leq \dots \leq S_N$
- Set $A_1 = S_1, A_3 = S_2, \dots$
  - More generally, set $A_{2k-1} = S_k$
- The above process used elements $S_1, S_2, \dots, S_M$, where $M = \lceil \frac{N}{2} \rceil$
- To fill the even positions, we do something similar. Set $A_2 = S_{M+1}, A_4 = S_{M+2}, \dots$
  - That is, set $S_{2k} = S_{M+k}$

Note that this process makes A_i and A_{i+1} be about $N/2$ positions apart (in sorted order) for every i, which is the best we can hope far.

This gives us a candidate zig-zag array A. Now check if it is valid (by looking at all size-3 subarrays), and if it is, print it.

If it isn't valid, use a similar process to create a zig-zag array based on the pattern $A_1 > A_2 < A_3 > A_4 < A_5 > \dots$
Notice that one easy way to do this is to simply repeat the above process on the reverse of S.

Once again, check if the obtained array is valid and print it if it is.

If both checks above fail, the answer is -1: every zig-zag rearrangement will include some pair of adjacent equal elements.

Proof

This is surprisingly a bit non-trivial to prove, and needs some case analysis.

Let's construct the array $A_1 < A_2 > A_3 < \ldots$ using the process as described above.
Suppose there are adjacent equal elements. Let i be the smallest index such that $A_i = A_{i+1}$.

The order in which we placed elements specifically tells us the following:

- $i = 2k$ for some $k \geq 1$
- $A_2 = A_4 = \ldots = A_{2k} = A_{2k+1} = A_{2k+3} = \ldots$

Let this element be x. The above tells us that x appears at least $\lceil \frac{N}{2} \rceil$ times in A.
If it appears $> \lceil \frac{N}{2} \rceil$ times, the pigeonhole principle tells us that there will always be some adjacent equal elements, so the answer will always be -1.

Now, suppose x appears exactly $\lceil \frac{N}{2} \rceil$ times. There are now two cases to consider, based on the parity of N.

- When N is odd, the only possible situation when a valid zig-zag array exists is when x is either the smallest or the largest element of A: these two cases correspond to the two constructions we had above.
- When N is even, a valid zig-zag array always exists because we can place elements in the form `x _ x _ x _ ... x _ _ x _ x ... _ x`, and choosing the spot where we flip from `x _` to `_ x` can be done based on how many elements are $< x$.
  - Once again, depending on whether x is the smallest element or not, you can see this corresponds to one of the constructions above.

This completes the proof.

TIME COMPLEXITY:

$\mathrm{O}(N \log N)$ per testcase.

PREREQUISITES:

Binary search, DFS

PROBLEM:

There is a tree with one node marked as hidden, which you want to find.

You can ask atmost 11 queries of the following form:

- Provide a set of edges to the judge. These edges are deleted from the tree. Then, the judge returns the maximum distance of some node from the marked node in the resulting forest.

Find the marked node.

EXPLANATION:

The constraints hint at binary search or some similar halving method, but how do we apply it?

There are in fact several different solutions, I will detail one of them below.

We initially have no information at all, i.e, the marked node could be any of the N nodes.
Ideally, after each query, we should be able to decrease the number of nodes that can possibly be the marked node by about half: doing this 18 times will leave us with a single candidate for the answer.

One way to do this is to *isolate* half the nodes. That is, a process like this:

- Let V be our current set of candidates. Initially, $V = \{1, 2, 3, \ldots, N\}$
- Delete some edges so that V splits into two sets V_1 and V_2 of equal size with the following property:
  - Every node in V_1 has **no** edge adjacent to it in the resulting forest
  - Every node in V_2 has **at least one** edge adjacent to it in the resulting forst

If we are able to do this, the the answer to the query gives us the following information:

- If the answer to the query is 0, then the marked node is isolated, so we can replace V with V_1.
- Otherwise, it is not isolated and we can replace V with V_2.

Now we just need to figure out a way to isolate half the vertices.

Isolation

One nice way to do this is to look at dfs (or bfs) order.
Let's perform a DFS from some node and then arrange the vertices in DFS order, say $v_1, v_2, \ldots, v_N$.
For $i \geq 2$, let e_i denote the index of the edge joining v_i to its parent.

Note that deleting e_N isolates v_N, deleting e_N and e_{N-1} isolates v_N and v_{N-1}, and so on.
In particular, deleting $e_i, e_{i+1}, e_{i+2}, \ldots, e_N$ isolates exactly the set of vertices $v_i, v_{i+1}, \ldots, v_N$.
(There is one caveat: if $i = 2$ then we delete every edge, so v_1 is also isolated. We'll come back to this later).

Notice that this allows us to convert our earlier halving process to just plain binary search!
That is, we can binary search on the DFS ordered edges, each time removing a suffix of edges, to find the first time the answer to a query is 0. This tells us which v_i is the marked node.

However, there is one catch, coming back to our earlier observation about $i = 2$: if we need to remove every edge to get the query to return 0, then the answer is either v_1 or v_2, so we need to be able to distinguish between them.

Luckily, the query limit of 18 allows us one query to do this.
Once again using the isolation idea, we just need to find a set of edges to remove that isolates one vertex but

not the other.

This is not hard to do. For example,

- Suppose $deg(v_1) \leq deg(v_2)$. Remove every edge adjacent to v_1: this isolates v_1 but not v_2
- If v_2 has lower degree, remove every edge adjacent to v_2 instead.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PROBLEM:

Shivansh and Tina play a game; Shivansh goes first.
They have a number with them, initially N. On their turn, a player should pick a non-negative integer power of K that doesn't exceed N, and subtract it from N.

Under optimal play, who wins?

EXPLANATION:

With two-player games like this, It often helps to analyze some small cases first, for example with $K = 2, 3, 4, \text{ or } 5$.

Some simple analysis reveals that when $K = 3$, Shivansh can always win if N is odd, and loses otherwise.
This is because 3^x is odd no matter what the value of x is. So:

- If N is odd, an odd number of turns will have passed when N reaches zero.
  This means Shivansh made the last move.
- If N is even, an even number of turns will have passed when N reaches zero.
  This means Tina made the last move.

In fact, the above analysis holds true for all odd K: Shivansh wins if N is odd, and loses otherwise.

For even K, let's look at what happens for small N.
Clearly, Shivansh can win if $N = 1, 3, 5, 7, \ldots, K - 1$ and Tina can win $N = 0, 2, 4, 6, 8, \ldots, K - 2$.
That's because for these values of N, the only possible move is to subtract 1; so the winner is decided by parity.
However, Shivansh can win $N = K$ by just subtracting K.

After this,

- Tina wins $N = K + 1$, since Shivansh must subtract either 1 or K and Tina subtracts the other one.
- Shivansh wins $N = K + 2$, by subtracting 1 and giving $N = K + 1$ (a losing state) to Tina.
- Tina wins $N = K + 3$: whether he subtracts K or 1, 3 and $K + 2$ are both winning states for Tina.
- Generally, Tina wins $K + 1, K + 3, K + 5, \ldots, 2K - 1$; and Shivansh wins $K + 2, K + 4, \ldots, 2K$
- However, Shivansh wins $2K + 1$; since he can subtract K on his turn and give $K + 1$ to Tina (which is losing, as noted above)

You might notice that this process is consistent in 'blocks' of length $K + 1$ starting from 0.
That is, in each such block, Tina wins the even positions and Shivansh wins the odd ones; except Shivansh wins the last position despite it being even.

This is indeed the pattern for Shivansh's win condition.
That is, let $r = N \pmod{K + 1}$, i.e, r is the remainder when N is divided by $K + 1$.
Then, Shivansh wins if r is odd or $r = K$, and loses otherwise.

Proof

We'll prove this by induction, though the basic idea was outlined above.

For the base case, we already know it's true for the segment $[0, K + 1)$, as noted above.
Now, assume it's true for $[0, x \cdot (K + 1))$, and we'll prove it's true for $[0, (x + 1) \cdot (K + 1))$.

The main observation required here is the fact that K^x is always either 1 or -1 modulo $(K + 1)$, which is easy to see (in fact, since $K \equiv -1 \pmod{K + 1}$, the powers of K alternate between 1 and -1).
So,

- $x \cdot (K+1)$ is a losing state: no matter what move is made, we move to either $1 \pmod{K+1}$ or $K \pmod{K+1}$, and in the range $[0, x \cdot (K+1))$; by the inductive hypothesis these are winning states for the opponent.
- $x \cdot (K+1) + 1$ is a winning state: subtract 1 to put the opponent in a losing state.
- $x \cdot (K+1) + 2$ is a losing state: either we subtract 1 and move to $(x \cdot (K+1) + 1)$, which is winning for the opponent; or we subtract move to an odd value modulo $K+1$ in the range $[0, x \cdot (K+1))$ which is still winning for the opponent.
- Extrapolating this, all odd numbers are winning and even numbers are losing.
- However, $x \cdot (K+1) + K$ is winning, since we can subtract K.

This takes care of all integers in the range $[x \cdot (K+1), (x+1) \cdot (K+1))$; and so we're done.

In fact, this pattern holds true for odd K as well, since we didn't really use the parity of K anywhere.

TIME COMPLEXITY

$\mathrm{O}(1)$ per test case.

PREREQUISITES:

Combinatorics

PROBLEM:

You have an array A of N integers. It's called K-beautiful if every subarray of size K has the same sum.

You can perform **at most** M moves, each one increasing some A_i by 1.
How many possible K-beautiful final arrays are possible?

EXPLANATION:

Suppose A was a K-beautiful array.
Then, the sum $(A_i + A_{i+1} + \ldots + A_{i+K-1})$ should be constant across all i.
In particular, this means $(A_i + A_{i+1} + \ldots + A_{i+K-1}) = (A_{i+1} + A_{i+2} + \ldots + A_{i+K})$, which tells us that $A_i = A_{i+K}$.

That is, if we fix the first K elements of A, the remaining values are fixed.

First however, we need to make everything equal to a baseline.
For each $1 \le i \le K$, let $M_i = \max(A_i, A_{i+K}, A_{i+2K}, \ldots)$.
We need to first bring all these values up to M_i, which takes $(A_i - M_i) + (A_{i+K} - M_i) + \ldots$ operations.
Compute this for each i and sum up the number of operations needed.

If this baseline number of operations is larger than M, then the answer is immediately 0 since no K-beautiful array can be constructed. Otherwise, remove these operations from M; so now we have upto M 'free' operations.

We are now ready to actually solve the problem.
Note that the final array is entirely determined by the (final) values of $A_1, A_2, \ldots, A_K$.
We know their current values, so the only thing that matters is how much we add to each index.

So, suppose we add x_i to index i.
Each time we add 1 to index i, we also need to add 1 to indices $i+K, i+2K, i+3K, \ldots$ in order to preserve the K-beauty condition.
In particular, suppose there are c_i such indices (including i).

The total number of operations we use is then $c_1x_1 + c_2x_2 + \ldots + c_Kx_K$.

Our objective is thus to count the number of solutions to

$$c_1x_1 + c_2x_2 + \ldots + c_Kx_K \le M$$

such that each x_i is ≥ 0.

A quadratic solution

Let's fix m $(0 \le m \le M)$ and count the number of solutions to $c_1x_1 + c_2x_2 + \ldots + c_Kx_K = m$.
This should remind you of the classical **stars-and-bars** problem, but it isn't quite in that form.

First, note that the c_i aren't arbitrary integers: in fact, each one of them is either $\lceil \frac{N}{K} \rceil$ or $\lfloor \frac{N}{K} \rfloor$, depending on i.

For now, let's assume K doesn't divide N so those two numbers are different.
Let $y = \lfloor \frac{N}{K} \rfloor$, and suppose r of the c_i are equal to y.
y and r are constants independent of m.

Rewriting the equation in terms of y and reducing it a bit, we see that

$$c_1x_1 + c_2x_2 + \ldots + c_Kx_K = m$$
$$y \cdot (x_1 + x_2 + \ldots + x_r) + (y+1) \cdot (x_{r+1} + x_{r+2} + \ldots + x_K) = m$$

Now, notice that $(x_1 + x_2 + \ldots + x_r)$ and $(x_{r+1} + x_{r+2} + \ldots + x_K)$ are two essentially independent summations.

Recall that we fixed m. Let's also fix $m_1 = x_1 + x_2 + \ldots + x_r$.
Notice that this uniquely fixes the value of $x_{r+1} + x_{r+2} + \ldots + x_K$, say to m_2.

The equations for m_1 and m_2 are in the stars-and-bars form, and so the number of solutions to each one can be computed in $\mathrm{O}(1)$ time.
The respective coefficients are $\binom{m_1 + r - 1}{m_1}$ and $\binom{m_2 + K - r - 1}{m_2}$.

Simply multiply these two numbers together to obtain the number of solutions for these fixed values of m and m_1.

Iterating over each (m, m_1) pair gives us a solution in $\mathrm{O}(M^2)$, which is too slow.

Note that if K divides N then fixing m also fixes $m_1 = \frac{m}{y}$ (and m_2 doesn't even come into the picture), so this case can be solved in $\mathrm{O}(M)$ time already.

Speeding up

Notice that fixing any two of m, m_1, m_2 fixes the third one uniquely.
So, let's fix just m_1 and see what we get.

If we also fix m_2, the resulting value of m must be $\leq M$.
In particular, m_2 must satisfy the inequality $y \cdot m_1 + (y+1) \cdot m_2 \leq M$.

This means that the valid values of m_2 form a consecutive set of integers starting from 0.
For each one, we need to add $\binom{m_1+r-1}{m_1}\binom{m_2+K-r-1}{m_2}$ to the answer.

Note that the first quantity is fixed since m_1 is fixed, so we just need the sum of $\binom{m_2+K-r-1}{m_2}$ across all valid m_2.
This isn't too hard: since the m_2 form a contiguous range starting from 0, we can create an array V where $V_i = \binom{i+K-r-1}{i}$ and take its prefix sums.

This allows us to process a single m_1 in $\mathrm{O}(1)$ time, so iterating it from 0 to M solves the problem in $\mathrm{O}(N+M)$ time.

TIME COMPLEXITY

$\mathrm{O}(N+M)$ per testcase.

PREREQUISITES:

Depth-first search, binary search (in the binary-lifting style)

PROBLEM:

You're given a tree on N vertices and an integer K.
The i-th vertex has value A_i written on it.

Split the tree into a forest of $K+1$ trees such that the bitwise XOR of the bitwise ANDs of the trees is maximized.

EXPLANATION:

As in many problems dealing with bitwise operations, we can build the answer bit-by-bit.

To maximize the answer, it's better to have a higher bit in the answer, even at the cost of lower bits.
So, lets iterate over bits from 29 down to 0, each time checking whether it can be added to the answer.

That is, we have the following algorithm:

- Initialize $ans = 0$.
- For each bit b from 29 to 0:
  - If it's possible for the answer to be at least $(ans + 2^b)$, set $ans \leftarrow ans + 2^b$.
  - Otherwise, do nothing.

The final value of ans is what we want.

Notice that this requires us to check whether the answer can be at least $ans + 2^b$.
Let's focus on doing that now.
For convenience, let $x = ans + 2^b$.

Notice that our process builds the answer bit-by-bit, so in our check, we can simply ignore all bits $< b$ (which are all unset in x anyway).
Also, every bit larger than b has already been fixed, so we can't change those.
In particular, if a bit larger than b is 0 in x, then we know there's no way it can be in the answer.
So, we can ignore this bit when performing computations.

Note that the above discussion tells us that we're ignoring all bits that are 0 in x.
In particular, that means we're actually considering the values of $A_i \ \& \ x$ for each i, instead of just A_i.

This means that we simply need to check whether we can divide the given tree into K trees such that:

- Let B_j be the bitwise XOR of the $(A_i \ \& \ x)$ values of the j-th tree.
- Then, we want $B_1 \ \& \ B_2 \ \& \ \ldots \ \& \ B_K = x$

Now, note that for x to be the bitwise AND of the B_j values, each of them must be a supermask of x.
However, each B_j is the bitwise XOR of some values that are submasks of x; thus making it also a submask of x.
Being both a submask and a supermask means that every B_j must be equal to x.

So, our objective is to figure out whether the tree can be partitioned into K trees, each with bitwise XOR x.

To do that, let's find out the *maximum* number of subtrees with XOR x that the given tree can be partitioned into.
That can be done using DFS, as follows:

- Start a DFS from some node, say 1.

- When at a node u,
  - Let S_u denote the subtree XOR of u.
  - If $S_u \neq x$, do nothing.
  - If $S_u = x$, we've found one subtree with XOR x. We can 'cut' it out, and then pretend the edge between u and its parent doesn't exist anymore (so u and its descendants won't contribute to the subtree XOR of the parent of u).

The number of subtrees we cut out is the maximum number we're looking for.
Let it be y.
Now,

- If $y < K$, then forming K trees is obviously impossible.
- Otherwise, $y \geq K$.
- Note that when $y \geq 3$, we can form $y - 2$ trees with XOR x by merging three adjacent trees. - In particular, repeating this allows us to form $y, y-2, y-4, \ldots$ trees, i.e, any number with the same parity as y.
- It's not hard to see that these are the only possible numbers achievable.
- So, obtaining K trees is possible if and only if $y \geq K$ and $(y - K)$ is even.

Thus we have an $\mathrm{O}(N)$ time algorithm to check whether the answer is at least x.
Doing this for each bit gives us a solution in $\mathrm{O}(30N)$.

It can be noted that the method we use to build the answer bit-by-bit is technically the exact same thing as binary searching on the range $[0, 2^{30})$.
However, the function we're searching on isn't monotonic; we only get the correct answer by utilizing the fact that we can work bit-by-bit.

In particular, this means that if you binary search on $[0, M)$ where M is not a power of 2, you most likely won't get AC.

TIME COMPLEXITY

$\mathrm{O}(N \log \max(A))$ per test case.

PROBLEM:

You have an array A and an integer K.
In one move, you can choose **exactly** K elements of A and increase their values by 1.
Is is possible to make A a palindrome?

EXPLANATION:

Let's get a trivial edge-case out of the way first:
If $K = N$ then we can only increase every element. In this case, if A isn't already a palindrome it's obviously impossible to turn it into one.
From now on, we'll assume $K < N$.

Our objective now is to make $A_i = A_{N+1-i}$ for every $1 \le i \le N$.
In particular, if $A_i < A_{N+1-i}$, we need to increase it by $(A_{N+1-i} - A_i)$.
After that's achieved, A_i and A_{N+1-i} need to be increased by equal amounts, to preserve their equality.

When either N or K is odd, it can be seen that it's always possible to make A a palindrome.

How?

First, say K is odd.
We have the following simple strategy:

- If $A_i = A_{N+1-i}$ for every i, we're done.
- Otherwise, pick an index i such that $A_i < A_{N+1-i}$
- Next, choose $\frac{K-1}{2}$ pairs of opposite indices that don't include $N + 1 - i$ (which is always possible since $K < N$), for a total of K indices chosen.
- Perform the operation on these K indices, and continue.

Each move brings A_i closer to A_{N+1-i}, while not breaking the 'palindrome-ness' of all other indices.
It's not hard to see that it'll finish in finite time, at which point we'll be done.

Next, say N is odd and K is even.
Once again we have a similar strategy:

- Pick an index i such that $A_i < A_{N+1-i}$
- Pick the middle index, which can be increased freely
- Finally, pick $\frac{K-2}{2}$ pairs of opposite indices excluding $N + 1 - i$ (once again, always possible since $K < N$) and perform the operation on these K.

This leaves us with the case when N is even, K is even, and $K < N$.
Note that:

- Since N is even, if the final array is a palindrome its sum will be even.
- Since K is even, each move increases the sum of the array by an even number.

Together, this tells us that if $sum(A)$ is odd, it can never be turned into a palindrome.

What if $sum(A)$ is even?
We can always make A into a palindrome!

Proof

This can be proved with a minor modification to our proofs odd K and N.

Consider the following process:

- If $A_i = A_{N+1-i}$ for every i, we're done.
- Otherwise, find an index i such that $A_i < A_{N+1-i}$.
- There are now two possible cases.
- If there exists *another* index j such that $A_j < A_{N+1-j}$, then pick j as well, then pick $\frac{K-2}{2}$ opposite pairs not including these two and perform the operation on them.
- If no such j exists, then the sum of the array being even means that $A_i + 2 \leq A_{N+1-i}$.
  - So, pick an arbitrary index j (that is not $N+1-i$), and $\frac{K-2}{2}$ more opposite pairs and perform the operation on them.
  - Next, pick $i, N+1-j$, and then $\frac{K-2}{2}$ more pairs to perform the operation.
  - This way we've increased A_i by 2 while preserving 'palindrome-ness' of the other indices.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Familiarity with bitwise OR

PROBLEM:

You're given an array A containing at least two distinct elements, and an integer X.

For an integer $0 \le Y \le X$, define the array B as $B_i = A_i \mid Y$.
Find the largest Y such that B consists of at least two distinct elements.

EXPLANATION:

Since the operation we're dealing with is bitwise OR, it helps to look at things bit-by-bit.

First, a useful definition: we say a bit b is *set* in the binary representation of integer x if the binary representation of x contains 2^b.
For example, $9 = (1001)_2 = 2^0 + 2^3$ has bits 0 and 3 set, while bits $1, 2, 4, 5, 6, \ldots$ are not set.

Suppose we pick a Y and compute the array B, and there are two unequal elements; say $B_i \neq B_j$.
Then, in particular the binary representation of B_i and B_j must differ.
That is, there must exist a bit b which is set in the binary representation of B_i but not in B_j, or vice versa.

Without loss of generality, let's say b is set in B_i and unset in B_j.
Then, we can observe the following:

- Y cannot have bit b set (otherwise it would be set in $B_j = A_j \mid Y$)
- This means A_i must've had bit b set, and A_j must've had it unset.

This tells us that for any valid Y, there will exist a bit b such that:

- b is unset in Y,
- Some element of A has this bit set, and
- Some element of A has this bit unset

Conversely, any Y that has such a bit b can easily be seen to satisfy the condition.

So, let's fix the value of b and try to find the largest Y we can with respect to it; call this Y_b.
The answer is the maximum value of Y_b across all b.

Note that if b is such that:

- Every element of A has b set; or
- Every element of A has b unset

then b must be ignored, as per our rules above.
For a fixed b, this can easily be checked in $\mathrm{O}(N)$ by iterating across all the elements.

Now the question is, how to find Y_b for a fixed b?
That can be done greedily.

- Iterate over bits from largest to smallest, i.e, from 29 down to 0.
- Initially, let $Y_b = 0$.
- Then, when processing bit i,
  - If $i = b$, do nothing. This will ensure that Y_b doesn't have bit b set, as required.

- Otherwise, if $Y_b + 2^i \le X$, add 2^i to Y_b.
  This way, we ensure that we never exceed X, while also making Y_b as large as possible.

So, for each b:

- We first check whether it is possibly valid, i.e, some element of A has it set and some other element has it unset.
  This is done in $\mathrm{O}(N)$.
- If it's valid, we construct Y_b using the above algorithm. This is done in $\mathrm{O}(30)$.

Since we do this for each bit from 0 to 29, the overall runtime is $\mathrm{O}(30 \cdot N + 30^2)$ which is good enough.

TIME COMPLEXITY

$\mathrm{O}(30 \cdot N + 30^2)$ per test case.

PROBLEM:

You're given a string S consisting of lowercase Latin characters.
Find a bracket sequence T of the same length as S such that:

- T is a regular bracket sequence
- If $S_i = S_{i+1}$, then $T_i = T_{i+1}$

EXPLANATION:

The condition $S_i = S_{i+1} \implies T_i = T_{i+1}$ means that each contiguous block of equal characters in S must correspond to a contiguous block of either `(` or `)`, of the same length, in T.

So, let's find all the maximal contiguous blocks of equal characters in S.
Suppose there are k such blocks, and their lengths are $L_1, L_2, \ldots, L_k$ in order.
For example, if $S = \text{aabacccbbddd}$ then $L = [2, 1, 1, 3, 2, 3]$.

Now, for each of these blocks, we need to decide whether it corresponds to `(` or `)`.
There are 2^k possibilities, and since k can be as large as N, checking each one is out of the question.

Instead, note that T is a regular bracket sequence iff:

- Each prefix of T has a non-negative balance.
- T as a whole has balance 0.

Here, the balance of a bracket sequence equals the number of opening parentheses it contains, minus the number of its closing parentheses.
For example, the balance of $()$ and $)($ are both 0, while $)(()($ has balance $3 - 2 = 1$.

This allows us to only have to deal with prefixes; in particular, we only really care about the balance of each prefix.
This reduction lets us use dynamic programming.

Let $\text{dp}[i][j]$ be a boolean value: `True` if there is a way to assign `(` and `)` to the first i blocks such that resulting balance is exactly j; and false otherwise.
Note that we only care about $j \geq 0$, since the balance cannot be allowed to go negative.

With this definition, transitions are in fact quite straightforward.
There are two choices for the i-th block: `(` or `)`. Trying each one,

- If the i-th block is turned into `(`, it will increase the balance by exactly A_i.
  So, the part before it must have a balance of $j - A_i$, and we can check whether this is possible by looking at $\text{dp}[i][j - A_i]$
- Similarly, if the i-th block is turned into `)`, we need to look at $\text{dp}[i-1][j + A_i]$.
- So, we have $\text{dp}[i][j] = \text{dp}[i-1][j - A_i] \mid \text{dp}[i-1][j + A_i]$, where $\mid$ denotes the bitwise OR operator.

Of course, the final answer is `Yes` if $\text{dp}[k][0]$ is `True`, and `No` otherwise; because the overall balance after all k blocks are assigned, should be 0.

If a solution exists, reconstructing it is not hard: move from the last block to the first, and each time the stored dp values will tell you which transition you took; and hence which type of bracket to assign to the current block.

There are upto $\text{O}(N^2)$ states and $\text{O}(1)$ transitions for each, for $\text{O}(N^2)$ time overall.

TIME COMPLEXITY

$\mathrm{O}(N^2)$ per testcase.

PROBLEM:

You are given integers A and B. Find the smallest possible value of $\text{lcm}(A, X) - \gcd(B, X)$ across all positive integers X.

EXPLANATION:

The answer is probably guessable by looking at the sample tests and/or working out a few small examples.

Short answer

The answer is simply

$$A - \gcd(A, B)$$

Long answer

I'll focus a bit on how one can derive this without guessing.

Our expression is $\text{lcm}(A, X) - \gcd(B, X)$.
In particular, the first part of that expression is always a multiple of A.

Let's fix the value of this lcm: suppose we choose X such that $\text{lcm}(A, X) = k \cdot A$.
Notice that this means that X is a factor of $k \cdot A$.

Our objective is now to *maximize* $\gcd(B, X)$, since that's what would make the difference as small as possible.
For $\gcd(B, X)$ to be as large as possible, X should contain as many prime factors in common with B as possible.

Along with the constraint that X is a factor of $k \cdot A$, this tells us that the best choice of X is $k \cdot A$ itself; choosing a strict factor isn't going to help us increase the gcd.

Ok, so now we have $k \cdot A - \gcd(B, k \cdot A)$.
What value of k minimizes this?

Intuitively, $k = 1$ seems good; after all, the second part is a factor of B so will always be $\leq B$ no matter how large k is; while the first part keeps increasing without bound.
$k = 1$ gives us $A - \gcd(A, B)$, which if you notice is the expression given in the "short answer" above.

Let's try to prove that this is optimal.

Proof

First, notice that for any $k \geq 1$, we have $\gcd(B, k \cdot A) \leq \gcd(k \cdot B, k \cdot A)$; since $k \cdot B$ is a multiple of B.

But it's also true that $\gcd(k \cdot B, k \cdot A) = k \cdot \gcd(A, B)$.
So, we have $\gcd(B, k \cdot A) \leq k \cdot \gcd(A, B)$.

This means that

$$\begin{aligned} k \cdot A - \gcd(B, k \cdot A) &\geq k \cdot A - k \cdot \gcd(A, B) \\ k \cdot A - \gcd(B, k \cdot A) &\geq k \cdot (A - \gcd(A, B)) \\ k \cdot A - \gcd(B, k \cdot A) &\geq A - \gcd(A, B) \end{aligned}$$

which is exactly what we set out to prove!

All that remains is computing the answer quickly, which requires us to compute $\gcd(A, B)$ quickly. This can be done in $\mathrm{O}(\log \min(A, B))$ using the **Euclidean algorithm**.

TIME COMPLEXITY:

$\mathrm{O}(\log \min(A, B))$ per testcase.

PROBLEM:

There is a hidden array A of length N. Find it using upto $3N$ queries of the following types:

- Given i and j, you will receive $\gcd(A_i, A_j)$
- Given i, j, k, you will receive $\text{lcm}(A_i, A_j, A_k)$

EXPLANATION:

With $3N$ operations, we need to use about 3 per index.

Note that constraints have $N \geq 4$, so we certainly need to be able to solve for $N = 4$. Let's attempt to do that first.

Solving $N = 4$

There are 10 possible queries we can ask on these four elements — 6 pairwise GCDs, and 4 triplet LCMs. Let's ask them all.

Let $g_{ij} = \gcd(A_i, A_j)$, and $L_{ijk} = \text{lcm}(A_i, A_j, A_k)$.

Recall that for two numbers x and y, we have the relation $\gcd(x, y) \cdot \text{lcm}(x, y) = x \cdot y$.
However, we now have the lcm of three numbers, so we'd like to generalize this a bit.
With a little understanding of how the above formula comes about, that's not very hard.

Details

For three numbers x, y, z, we want some relation between $\text{lcm}(x, y, z)$ and their gcds.
This can be done using the inclusion-exclusion principle.

Let's start with $\text{lcm}(x, y, z) = x \cdot y \cdot z$.
This is an overestimation, since any factor common to two of them has been multiplied twice.
So, we can divide out $\gcd(x, y)$, $\gcd(x, z)$, and $\gcd(y, z)$.

However, this is now an underestimation: any factor that's common to all three has been taken out too many times, so we add it back in, i.e, multiply by $\gcd(x, y, z)$.

This gives us

$$\text{lcm}(x, y, z) = \frac{x \cdot y \cdot z \cdot \gcd(x, y, z)}{\gcd(x, y)\gcd(x, z)\gcd(y, z)}$$

Note that this isn't quite a formal proof: why taking gcds is necessary and sufficient to cover the common factors takes some getting used to.
A more basic proof is to apply this argument to each prime factor of the LCM, then take the product of all the resulting prime powers.

You might notice that the formula we derived is pretty close to what we want.
For example, we have

$$L_{123} = \frac{A_1 \cdot A_2 \cdot A_3 \cdot \gcd(A_1, A_2, A_3)}{g_{12} \cdot g_{23} \cdot g_{13}}$$

We know most of the parts of this equation: everything except A_1, A_2, A_3 in fact (since $\gcd(A_1, A_2, A_3) = \gcd(g_{12}, g_{13})$).
In particular, this tells us the value of the product $A_1 \cdot A_2 \cdot A_3$. Let this value be P_{123}

Similarly, we can obtain $P_{124}, P_{134}, P_{234}$.

Multiplying all these together, we have

$$P_{123} \times P_{124} \times P_{134} \times P_{234} = (A_1 \cdot A_2 \cdot A_3 \cdot A_4)^3$$

So, taking the cube root of their product gives us the value of $A_1 \cdot A_2 \cdot A_3 \cdot A_4$; from which computing the individual values of A_1, A_2, A_3, A_4 is quite easy, since we also know the products of all their triplets.

This solves $N = 4$ entirely, so let's move on to a full solution.

Solving $N > 4$

Let's first use the $N = 4$ solution to find the first 4 elements.

Suppose we want to find A_i, where $i > 4$.
Recall the lcm-gcd relation we had from earlier. In particular, applying it to indices $(1, 2, i)$ gives us

$$L_{12i} = \frac{A_1 \cdot A_2 \cdot A_i \cdot \gcd(A_1, A_2, A_i)}{g_{12} \cdot g_{2i} \cdot g_{1i}}$$

However, we already know A_1 and A_2, so the only new pieces of information we require here are L_{12i}, g_{1i}, and g_{2i}; each of which require one query.
This also allows us to compute $\gcd(A_1, A_2, A_i)$, after which computing A_i is easy since we know every other part of that expression.

This takes 3 queries for each index past the fourth, and 10 queries for the first 4, for a total of $3N - 2$ queries.

Implementation and large numbers

The above solution idea did not go into specifics about the intermediate values, but the observant reader may notice that we deal with some rather large numbers.
In particular, $(A_1 \cdot A_2 \cdot A_3 \cdot A_4)$ can be as large as 10^{72}, which definitely doesn't fit in the limit of `long long`.

There are a few ways to get around this.
By far the simplest is to simply use a language with support for large numbers, such as Python or Java.
In C++, the simplest way is probably to use a `BigInteger` class pulled from somewhere.

If you're not willing to do either of the above, there is still hope.
First off, notice that while the numbers we deal with can get quite large, they are products of numbers that are $\leq 10^6$.
This means that all their prime factors are $\leq 10^6$.

Since multiplication and division can be simulated easily on the prime factorizations of numbers, this gives us one way of dealing with the issue.

Once you know $A_1 \cdot A_2 \cdot A_3$, prime factorize it by iterating over all numbers from 1 to 10^6. Note that iterating till $\sqrt{A_1 \cdot A_2 \cdot A_3}$ as usual might TLE.
Similarly, prime factorize the other three product triples, which allows you to compute the prime factorization of $A_1 \cdot A_2 \cdot A_3 \cdot A_4$. After this, each of the four elements can be computed from this prime factorization, and we're done.

Note that every other value does in fact fit inside 10^{18}, as long as you take care of the order of operations correctly (i.e, divide first before multiplying when possible).

A small challenge

It's possible to solve this problem in C++ without using a `BigInteger` class and without prime factorizations (which would allow you to solve for, say $T \leq 10^4$ and $sum(N) \leq 10^4$).

In particular, only the use of `128`-bit integers is necessary, i.e, the `__int128_t` data type. Do you see a way to do this?

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

DFS, finding a tree's centroid

PROBLEM:

You're given a tree on N vertices.
For a fixed root r and permutation P of the vertices, the score of the tree is defined to be the size of the set $\{\text{lca}(P_i, P_{i+1}) \mid 1 \le i < N\}$.

Find the minimum possible score, and also values of r and P that attain this minimum.

EXPLANATION:

For a fixed root r and permutation P of the vertices, let $S(r, P)$ denote the set $\{\text{lca}(P_i, P_{i+1}) \mid 1 \le i < N\}$, and $|S(r, P)|$ denote its size.

The best we can hope to do is to achieve a score of 1, so let's check if that's possible.

Suppose we root the tree at r.
Then, $\text{lca}(r, x) = r$ for any vertex x, so no matter which P we choose, the set $S(r, P)$ will contain r.
Thus, if we want $|S(r, P)| = 1$, we have to check whether there exists a permutation P such that $\text{lca}(P_i, P_{i+1}) = r$ for *every* i.

Let the children of r be vertices $c_1, c_2, \ldots, c_k$, and their corresponding subtrees be $T_1, T_2, \ldots, T_k$.
Note that if two adjacent elements of P are from the same T_i, then their lca can never be r.
So, we need to check whether there exists an ordering such that adjacent vertices are from distinct subtrees.

Intuitively, if one of the subtrees is too large, this is not possible.
In particular, if some subtree has size $> \lceil \frac{N}{2} \rceil$, then any ordering will have some two vertices of this subtree adjacent to each other.

What about the case when all the subtrees have sizes $\le \lceil \frac{N}{2} \rceil$?
In fact, we can always find a valid ordering in that case!

There are many constructions, here's one.
Let's sort the subtrees $T_1, T_2, \ldots, T_k$ in descending order of size, i.e, $size(T_i) \ge size(T_{i+1})$.

We'll place all vertices from T_1 first, then T_2, then T_3, and so on.
To ensure that no two from the same subtree are adjacent, place vertices at positions $1, 3, 5, 7, \ldots$ followed by $2, 4, 6, 8, \ldots$

That is, fill the odd positions first, and then fill the even positions.

Of course, this construction hinges on the fact that all children of the root have subtree sizes not exceeding $\frac{N}{2}$, so we need to find such a root r first.

The simplest way to do this is to choose a vertex that minimizes the value of its maximum child subtree size — it can be proved that for such a vertex, the maximum subtree size will be $\le \lfloor \frac{N}{2} \rfloor$.
Such a vertex is also called a centroid of the tree, and can easily be found in $\text{O}(N)$. See the first section of **this page**, for example.

TIME COMPLEXITY

$\text{O}(N \log N)$ per test case.

PREREQUISITES:

Prefix sums

PROBLEM:

You are given an array A. Find the largest possible value of $r - l + 1$ such that $A_l \mid A_{l+1} \mid \ldots \mid A_r = A_1 \mid A_2 \mid \ldots \mid A_{l-1} \mid A_{r+1} \mid \ldots \mid A_N$.

Here, $\mid$ denotes bitwise OR.

EXPLANATION:

Let $M = A_1 \mid \ldots \mid A_N$ be the bitwise OR of all the elements.

Let's call subarray $[l, r]$ *valid* if its bitwise OR equals the bitwise OR of the rest of the array.

Note that the bitwise OR of any valid subarray *must* be M.

Why?

Let the bitwise OR of the subarray be x. Then, the outside also has bitwise OR x, and together they make up the whole array.
So, $x \mid x = M$, but $x \mid x = x$ so we have $x = M$.

To phrase this differently, a subarray is valid if and only if, for each bit b that occurs in M,

- $A[l, r]$ contains an element with the b-th bit set; **and**
- $A[1, l-1] \cup A[r+1, N]$ contains an element with the b-th bit set.

Let's use this fact.
Suppose we fix a position l. Let's try to find the largest r such that $[l, r]$ is valid: if we can do this for every l, the answer is the maximum value of $r - l + 1$ among them all.

Now that l is fixed, let's look at a specific bit b that is present in M.

- If b is present somewhere in the subarray $[1, l-1]$, then the 'outside' is good with respect to this bit.
- Otherwise, we need to ensure that b is present **after** r.
- In particular, if $last_b$ is the last position that contains b, then we must have $r < last_b$.

Each b that is not present in the prefix gives us one such constraint on r. Apply all of them to obtain the maximum possible r we can get if we start at l.

Now, the outside is taken care of, so we need to check the inside.
That is, for each b, we need to check whether it appears in the range $[l, r]$ or not.

Putting everything together, we require the following:

- Given l and b, check whether $[0, l-1]$ contains b
- Given l, r, b, check whether $[l, r]$ contains b
- Given b, find $last_b$, the last index at which b appears

The $last$ array in the third part can be precomputed by iterating across the array and just looking at each bit.

The first and second parts can be solved with the help of prefix sums.
Let $P(i, b)$ denote the number of times b appears in $[1, i]$. Note that if we knew every $P(i, b)$, then:

- Checking whether b lies in $[0, i-1]$ is the same as checking whether $P(i-1, b) > 0$
- Checking whether b lies in $[l, r]$ is the same as checking whether $P(r, b) - P(l-1, b) > 0$

$P(i, b)$ can be calculated as follows:

- If A_i contains the bit b, then $P(i, b) = P(i-1, b) + 1$
- Otherwise, $P(i, b) = P(i-1, b)$

This allows us to compute every $P(i, b)$ in $\mathrm{O}(30N)$ time, and with this the rest of the problem can also be solved in $\mathrm{O}(30N)$ as described above.

TIME COMPLEXITY:

$\mathrm{O}(30 \cdot N)$ per testcase.

PROBLEM:

There are N shops; the i-th shop sells items with value V_i.
Each shop has an infinite supply of items, and each item costs exactly one dollar.

You can buy items worth at most K dollars. However, the following constraints must be followed:

- Suppose you but num_i items from the i-th shop.
- Then:
  - $\text{num}_1 = 0$ or $\text{num}_i = 1$
  - For $i > 1$, $\text{num}_i \le \text{num}_{i-1} + 1$

Find the maximum total value of items you can buy.

EXPLANATION:

With bounds on the number of items you can take and an objective of maximizing values, this problem seems somewhat similar to the knapsack problem.
And indeed, this task can indeed be solved with dynamic programming, with each subtask requiring a slightly higher level of optimization.

Subtask 1

Let's start with the easiest one.

A very natural dp state should encapsulate:

- Which shop we're at, say i.
- The total number of items we've bought so far, say x.
- num_i, the number of items we're buying from this shop.

So, let $dp(i, x, y)$ be the maximum value we can obtain if we bought exactly x items from the first i shops, and of them, bought exactly y items from the i-th shop.

Buying y items from the i-th shop gives us a value of $y \cdot V_i$.
Further, we must've bought exactly $x - y$ items from the previous shops; and in also *at least* $y - 1$ items from shop $i - 1$.

This gives us the transitions:

$$dp(i, x, y) = y \cdot V_i + \max(dp(i - 1, x - y, z))$$

where the maximum is across all $y - 1 \le z \le K$.

There are $\text{O}(NK^2)$ dp states here (i is upto N, x and y are upto K)2 and the transition from each one can be done in $\text{O}(K)$ time for an overall $\text{O}(NK^3)$ solution.

Subtask 2

Now N and K are a bit higher, so the previous solution is too slow.

Let's look at the dp transitions.
We have

$$dp(i, x, y) = y \cdot V_i + \max(dp(i - 1, x - y, z))$$

across $y - 1 \le z$.
The slow part is computing the maximum across all z.

Note that when i, x, y are fixed, what we're looking for is a *suffix maximum* of the $dp(i-1, x-y, \cdot)$ table.
So, let's just calculate these suffix maximums as well!

Define $\text{suff}(i, x, y) = \max(dp(i, x, y), dp(i, x, y+1), dp(i, x, y+2), \ldots, dp(i, x, K))$.
Then, $\text{suff}(i, x, y) = \max(dp(i, x, y), \text{suff}(i, x, y+1))$; so these suffix maximums can be computed as and when we compute the dp values.

With this in hand, we have $dp(i, x, y) = y \cdot V_i + \text{suff}(i-1, x-y, y-1)$.

So, there are now $\text{O}(NK^2)$ states, and an $\text{O}(1)$ transition from each.
This results in a $\text{O}(NK^2)$ solution, which is good enough for this subtask.
Subtask 3

N and K are now large enough that we can't even have NK^2 states anymore; it'll take too much memory and generally be too slow.

Let's try to cut it down a bit.
Suppose we buy y items from a shop.
Since we can increase the maximum by only a step each time, this means we must've bought $1, 2, 3, \ldots, y-1$ items from other shops before this.

In particular, if we buy y items from a shop, we've bought a total of *at least* $\frac{y \cdot (y+1)}{2}$ items in total.
We're only allowed to buy K items, so we don't really care about very large y.
In particular, we don't care about any y such that $\frac{y \cdot (y+1)}{2} > K$; which limits us to about $y \le \sqrt{2K}$.

In other words, we only care about $dp(i, x, y)$ for about $\text{O}(NK\sqrt{K})$ states; each of which has a $\text{O}(1)$ transition using the suffix trick from the previous subtask.

Implementing this optimization is enough to get AC.

Note that depending on your implementation, a solution with correct time complexity for subtasks 2 and 3 might be inexplicably much slower than other similar solutions.
This is most likely to be a memory-related issue.

You can optimize the memory requirement down from $\text{O}(NK^2)$ or $\text{O}(NK\sqrt{K})$ down to $\text{O}(K^2)$ or $\text{O}(K\sqrt{K})$ (or even $\text{O}(K)$ using a slightly different approach) by noting that $dp(i, \ldots)$ depends only on $dp(i-1, \ldots)$ — so, it's enough to keep only the previous row of the dp table, instead of all of them.

This is (maybe) not *required* to get AC, though you should likely see a major speedup on using it.
In slower languages like Python, it might be necessary to get AC.

TIME COMPLEXITY

$\text{O}(NK\sqrt{K})$ per test case.

PROBLEM:

You're given an undirected graph on N vertices with M edges.
K mail carriers are present on it: the i-th of them starts at vertex X_i and has a range of D_i; meaning he can deliver mail to any vertex that's at most distance D_i away from X_i.

Can every vertex receive its mail?

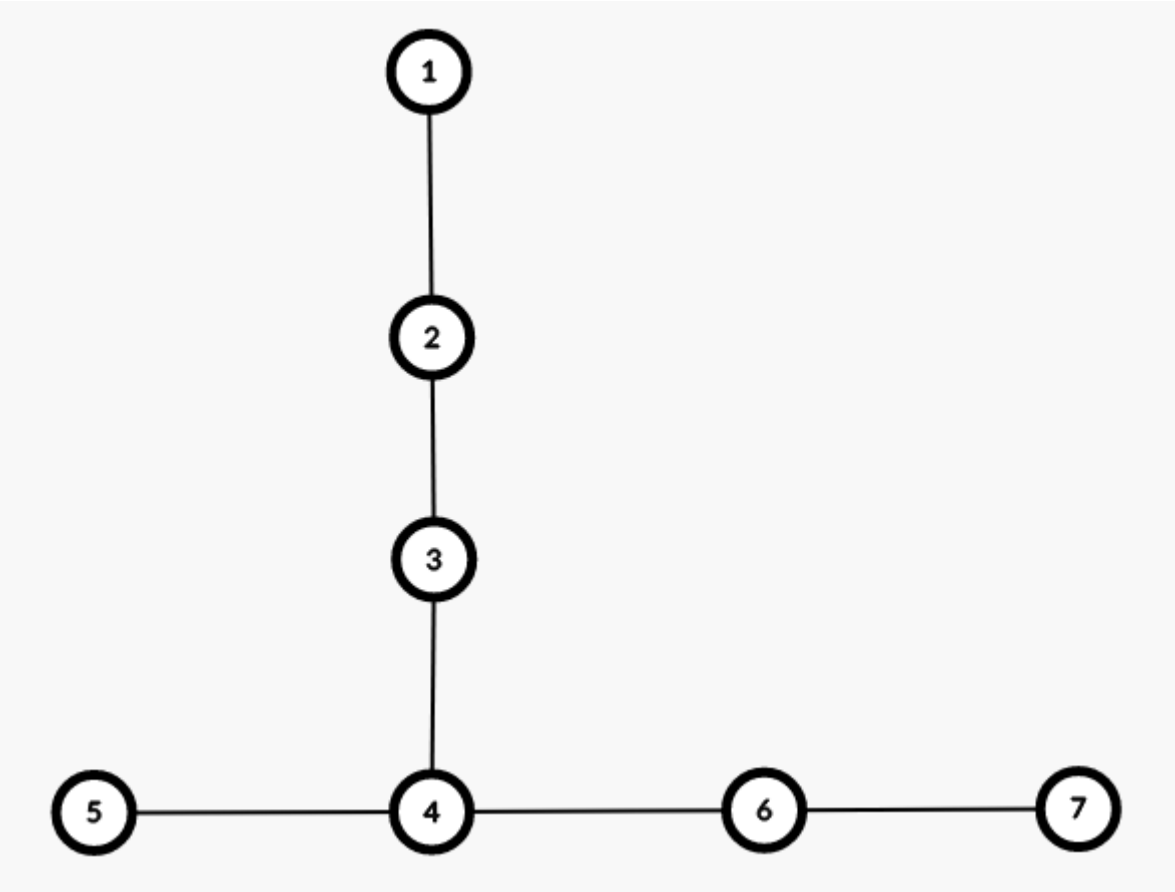
EXPLANATION:

If $K = 1$, we could run a BFS from the starting point and till distance D_i; and check if all vertices are covered in $\mathrm{O}(N + M)$.

This of course generalizes to a (slow) solution for $K > 1$: run a separate BFS for each mailman and check if all vertices are visited in the end.
Of course, the complexity here is $\mathrm{O}(K \cdot (N + M))$, which is too slow.

A natural optimization to try now would be to attempt a **multisource BFS** , however it doesn't quite work directly because of the D_i values; so a previously visited vertex can't simply be skipped.

Example



Suppose we have mail carriers at 1 and 5, with $D_1 = 5$ and $D_5 = 4$.

If we BFS from 1 first, we'll end up marking vertices 4 and 6, but not 7.
BFS-ing from 5 will then stop before marking 7, since vertices on its path are already marked.

This case also shows that trying to BFS from vertices with larger D_i first isn't necessarily correct either.
What really matters is how many moves we have left from each vertex.

In particular, we need to know the *maximum* number of remaining steps from each vertex.
To do this, we can instead transform our algorithm to a multisource *dijkstra*!

Let $\text{rem}[u]$ be the maximum number of remaining steps from vertex u.
First, for each mail carrier i, set $\text{rem}[K_i] \leftarrow \max(\text{rem}[K_i], D_i)$.

Then, for a vertex u and an adjacent vertex v, $\text{rem}[v]$ can be updated with $\text{rem}[u] - 1$.
Notice that this is similar to the usual dijkstra with distances, where we update $\text{dist}[v]$ with $\text{dist}[u] + wt$.

So, simply running dijkstra's algorithm on this will give us the correct values of $\text{rem}[u]$ for all u.
Note that here, we want to keep the *maximum* value and not the minimum; make sure to implement that correctly.

Finally, if $\text{rem}[u] \leq 0$ for any vertex the answer is `No`; otherwise the answer is `Yes`.

TIME COMPLEXITY

$\text{O}((N + M) \log N)$ for each testcase.

PREREQUISITES:

Familiarity with bitwise operations

PROBLEM:

You have an array A and an integer X. In one move, you can pick $1 \le i < j \le N$ and set both A_i and A_j to $(A_i \oplus A_j) \mid X$.

Find the minimum number of moves needed to make all the elements odd, or say that it's impossible.

EXPLANATION:

Looking at the bitwise operations we're dealing with:

- $(A_i \oplus A_j) \mid X$ is odd when at least one of X or $(A_i \oplus A_j)$ is odd.
- $(A_i \oplus A_j)$ is odd when A_i is odd and A_j is even, or vice versa.

In particular, this means that:

- If A_i is odd and A_j is even (or vice versa), in one move we can make them both odd.
- If A_i and A_j are both even, then
  - If X is even they'll remain even
  - If X is odd they'll both turn odd
- Operating on two odd numbers is obviously not optimal because we don't want to create more even numbers, so we don't need to consider it.

In summary, depending on the values of A_i, A_j, X we can either turn two evens into two odds, or one even into one odd.
Notice that the number of operations only really depends on the number of even and odd numbers in A.

So, let A have c_0 even numbers and c_1 odd numbers.
Now let's do a bit of casework:

- If $c_0 = 0$ then everything's already odd, so the answer is 0.
- If X is odd, we can turn 2 evens to 2 odds (or one even to one odd, if there's only one left). The minimum number of operations is thus $\lceil \frac{c_0}{2} \rceil$, where $\lceil \ \rceil$ denotes the ceiling function.
- This leaves us with the case when X is even.
  - If $c_1 = 0$, i.e there are no odd numbers, then the answer is -1; this is because the result of any operation will be even.
  - Otherwise, we can turn one even to one odd by operating on $(even, odd)$, so we need c_0 moves.

Computing c_0 and c_1 takes $\mathrm{O}(N)$ time, after which the casework is $\mathrm{O}(1)$.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

A *special permutation* of length N is a sequence of N non-negative integers that contains every integer in $[L, L+N-1]$ exactly once.

You have a permutation of length N.
An integer x is called valid if $[A_1 \oplus x, A_2 \oplus x, \ldots, A_N \oplus x]$ is a special permutation of length N.

Answer Q queries. In each query, you're given L and R — find the number of valid integers in the range $[L, R]$.

EXPLANATION:

N is way too large for anything other than $\mathrm{O}(\log N)$ or $\mathrm{O}(1)$ per query to be viable.
In counting problems like this, often it is useful to be able to nicely characterize the items we're counting — that is, we should first attempt to figure out what a *valid* x looks like.

Given that the operation at hand is bitwise XOR, it makes sense to look at binary representations.

So, suppose N is a k-bit integer.
Then, it's not hard to notice that:

- If x is such that *all* of its last k bits are 0, then x is valid.
  This is because for such an x, we have $x \oplus i = x + i$ for $i \le N$.
  So, the sequence we end up with is $[x+1, x+2, \ldots, x+N]$ which is a special permutation.
- If x is such that *all* of its last k bits are 1, again x is valid.
  For such an x, we have $x \oplus i = x - i$ for $i \le N$; once again giving us a special permutation.

In fact, these are the only valid types of x.

Proof

Now, given L and R, we need to count for each of the two types above, how many of them lie in range $[L, R]$.
That's not too hard with the following observations:

- The last k bits of x are 0 if and only if x is a multiple of 2^k.
  The number of multiples of 2^k in $[L, R]$ can be found using the formula $\lfloor \frac{R}{2^k} \rfloor - \lfloor \frac{L-1}{2^k} \rfloor$
  This follows from the fact that for any a, b, there are $\lfloor \frac{a}{b} \rfloor$ multiples of b that are $\le a$.
- The last k bits of x are 1 if and only if $x+1$ is a multiple of 2^k.
  To count such numbers, we can instead just count multiples of 2^k in the range $[L+1, R+1]$ using the above formula.

So, knowing k, each query can be answered in $\mathrm{O}(1)$ time.
Finding k is easy in $\mathrm{O}(\log N)$: just find the smallest integer k such that $2^k > N$.

TIME COMPLEXITY

$\mathrm{O}(\log N + Q)$ per testcase.

PROBLEM:

There's an $N \times M$ matrix with the value $(i-1)M+j$ written in cell (i,j).

Q updates are performed on this matrix, each update either multiplies all values in a row or a column by a certain number.
FInd the sum of the matrix after all operations are performed.

EXPLANATION:

This problem can be solved by writing out appropriate equations to simplify things.

First, let's calculate the sum of the matrix before any operations are performed.
This is just the sum of the first $N \times M$ integers, which is easy to compute (watch out for overflow though!).

Now, let's look at how updates affect this.
Suppose we multiply row r by x.
Then,

- The original sum of this row was $(r-1)M^2 + \frac{M(M+1)}{2}$.
- The new sum of this row is simply this value multiplied by x, since multiplying everything by x also multiplies their sum by x.
- So, we can update the sum in $\mathrm{O}(1)$.

A similar calculation can be done for column updates by finding a formula for column sums.

However, this doesn't quite take care of overlaps correctly.
For example, consider cell (i,j) (with initial value $(i-1)M+j$) such that row i was multiplied by value x_i and column j was multiplied by value y_j.
In our above scheme, we've independently increased the value of this cell for each update. That is, the row operation increased its value x_i times, and the column operation increased its value y_j times, for a total increase of $(x_i + y_j)$ times.

However, updates combine multiplicatively, so its real final value should be $(x_i \cdot y_j)$ times its initial value.

If we know i and j (and hence x_i and y_j), this is easy to fix. Unfortunately, there can be $\mathrm{O}(K^2)$ such affected cells (one for each pair of (row update, column update)), and we surely can't go through them all one by one.

To optimize this process, we can once again use a bit of math.

Suppose the column updates were to columns $c_1, c_2, \ldots, c_w$, with respective multipliers $y_1, y_2, \ldots, y_w$.
Let's look at a row update, say to row r with value x. We can process all the column updates to this row quickly.

How?

The set of affected cells are $(r, c_1), (r, c_2), \ldots, (r, c_w)$.

Based on our initial scheme, their current values are
$((r-1)M + c_1) \cdot (x + y_1)$
$((r-1)M + c_2) \cdot (x + y_2)$
$((r-1)M + c_3) \cdot (x + y_3)$

$\vdots$
$((r-1)M + c_w) \cdot (x + y_w)$

Expanding the parentheses, it can be seen that the sum of these values is
$(r-1)Mwx + x \cdot sum(c_i) + (r-1)M \cdot sum(y_i) + sum(c_i \cdot y_i)$

Note that $sum(c_i), sum(y_i), sum(c_i \cdot y_i)$ are constants independent of the row we choose, and so their values can be precomputed.
Knowing these values, along with r and x, this sum can be computed in $\mathrm{O}(1)$.

Now, let's look at what the values of these cells *should* be:
$((r-1)M + c_1) \cdot (x \cdot y_1)$
$((r-1)M + c_2) \cdot (x \cdot y_2)$
$((r-1)M + c_3) \cdot (x \cdot y_3)$

$\vdots$
$((r-1)M + c_w) \cdot (x \cdot y_w)$

Once again, this sum can be expanded, to obtain
$((r-1)Mx \cdot sum(y_i)) + x \cdot sum(c_i \cdot y_i)$

Just as above, this can be computed in $\mathrm{O}(1)$ time.

Since both the wrong and correct contributions can be computed in $\mathrm{O}(1)$ time, the entire row can be updated in $\mathrm{O}(1)$ time and we're done.

Do this for each row update to solve the problem in $\mathrm{O}(K)$.

TIME COMPLEXITY

$\mathrm{O}(K)$ per test case.

PREREQUISITES:

Greedy

PROBLEM:

Given a binary string S, you can perform the following operations on it:

- Change a 0 to a 1
- Insert a 0 anywhere in the string

Find the maximum possible integer that can be formed after at most K operations.

EXPLANATION:

Let X be the decimal value of the string.
Let's see how the given operations affect the value of X.

- Suppose we change bit i from 0 to 1. This increases the value of X by 2^i.
- Suppose we insert a 0 after bit i. This doesn't change anything about bits $0, 1, 2, \ldots, i-1$, but shifts bits $i, i+1, \ldots$ to the left by one step, i.e, multiplies their values by 2.
  So, if Y is the total value of bits $\geq Y$, X will increase by exactly Y.

From this, it's easy to see that whenever we insert a 0, it's always optimal to do so at the end of the string, since it directly multiplies the value of X by 2.

So, we only need to decide when to insert a 0 and when to change a 0 into a 1.
In fact, this can be done greedily!

- If the leftmost character of S is 1, then insert a 0 at the end of S
- Otherwise, change the leftmost character of S to 1.

Each operation is easily performed in $\mathrm{O}(1)$, so the final string can be found in $\mathrm{O}(N+K)$.

TIME COMPLEXITY

$\mathrm{O}(N+K)$ per test case.

PROBLEM:

Given a binary string S, consider another binary string X such that $X_{i+1} = S_i \oplus X_i$ for each $1 \le i < N$.

Find the maximum number of ones such a string can have.

EXPLANATION:

Note that one X_1 is fixed, the rest of the string is uniquely determined, since:

- $X_2 = X_1 \oplus S_1$
- $X_3 = X_2 \oplus S_2$

$\vdots$

But X is a **binary** string, so there are only two choices for X_1 — it can be either 0 or 1.
Simply try both options, construct the respective strings, and see which one has the maximum number of ones.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Algebraic manipulation

PROBLEM:

You have an array $[1, 2, 3, \ldots, N]$. At most K times, you can swap any two of its elements. What is the maximum number of inversions the resulting array can have?

EXPLANATION:

The absolute best final array we can hope for is $[N, N-1, N-2, \ldots, 3, 2, 1]$, which has $\binom{N}{2}$ inversions.

To this end, the optimal sequence of swaps is as follows:

- Swap 1 and N
- Swap 2 and $N-1$
- Swap 3 and $N-2$

$\vdots$

Notice that we need only $N/2$ swaps to reach $[N, N-1, \ldots, 2, 1]$, so we can set $K \leftarrow \min(N/2, K)$.

Now, simulating the above process in $\mathrm{O}(K)$ is easy, but the constraints don't allow for it. Instead, let's analyze it a bit more.

We start out with 0 inversions.

- Swapping 1 and N gives us $2N-3$ inversions: $N-1$ with 1 and $N-1$ with N, with the pair $(1, N)$ being counted twice
- The same logic should tell you that swapping 2 and N gives us $2N-7$ more inversions
- Swapping 3 and $N-2$ gives us $2N-11$ inversions

$\vdots$

So, K swaps give us

$$(2N-3) + (2N-7) + (2N-11) + \ldots + (2N-4K+1)$$

inversions. This is the sum we want to compute.

Doing a bit of basic algebra, this is not hard:

$$\begin{aligned}(2N-3) + (2N-7) + (2N-11) + \ldots + (2N-4K+1) &= 2NK - (3 + 7 + 11 + \ldots + (4K-1)) \\ &= 2NK - (4 + 8 + \ldots + 4K - K) \\ &= 2NK - (4(1 + 2 + \ldots + K) - K) \\ &= 2NK - (2K(K+1) - K)\end{aligned}$$

which is an $\mathrm{O}(1)$ formula for the answer!

Of course, it's possible to derive different formulas with the help of different algebraic manipulations.

TIME COMPLEXITY:

$\mathrm{O}(1)$ per testcase.

PROBLEM:

Let $F(x, y, z) = (x \mid y) - (x \,\&\, z)$.

Given A, B, C, find the number of integers $0 \le x < 2^C$ such that $F(x, A, B)$ is maximum.

EXPLANATION:

Almost everything in this problem is a bitwise operation, except the subtraction in $F(x, y, z)$.
However, notice that $(x \mid y)$ will contain all the set bits of x, while $(x \,\&\, z)$ cannot contain any bits that aren't set in x.
So, $(x \,\&\, z)$ is a *submask* of $(x \mid y)$, and this means the subtraction operation is really just bitwise XOR.
That is, we can rewrite the function as $F(x, y, z) = (x \mid y) \oplus (x \,\&\, z)$.

Now that everything we're dealing with is bitwise operation, it helps to look at what's going on bit-by-bit.

Let's fix a bit k and see what happens.
Let x_k denote the value of the k-th bit of x (which is either 0 or 1). Similarly define $A_k, B_k, F_k(x, A, B)$.
We already know A_k and B_k, our aim is to find out what values x_k can possibly take.

We have $F_k(x, A, B) = (x_k \mid A_k) \oplus (x_k \,\&\, B_k)$, and we'd like to maximize $F_k(x, A, B)$.

There are 4 cases depending on their values:

- $A_k = B_k = 1$
  Here, $(x_k \mid A_k) = 1$, and $(x_k \,\&\, B_k) = x_k$.
  So, $F_k(x, A, B) = 1 \oplus x_k$, which is maximized when $x_k = 0$.
- $A_k = B_k = 0$
  Here, $(x_k \mid A_k) = x_k$, and $(x_k \,\&\, B_k) = 0$.
  So, $F_k(x, A, B) = x_k \oplus 0$, which is maximized when $x_k = 1$.
- $A_k = 1$ and $B_k = 0$
  Here, $F_k(x, A, B) = 1 \oplus 0 = 1$, and is independent of x_k. So, we can choose x_k to be either 0 or 1
- $A_k = 0$ and $B_k = 1$
  Here, $F_k(x, A, B) = x_k \oplus x_k = 0$, once again we can choose x_k freely.

Putting the above together:

- If $A_k = B_k$, x_k is fixed and we have no choice.
- If $A_k \neq B_k$, x_k can be chosen freely.

So, the answer is simply 2^d, where d is the number of bits where A and B differ in their binary representations.
d can be computed by iterating over each bit independently; or you can notice that it's just the number of set bits in $(A \oplus B)$ for an $\mathrm{O}(1)$ solution.

TIME COMPLEXITY

$\mathrm{O}(C)$ or $\mathrm{O}(1)$ per test case.

PROBLEM:

Given an array A of N integers, construct a simple directed strongly connected graph on N vertices such that:

- $A_i = A_j$ means the edges $i \to j$ and $j \to i$ cannot exist. Also, A_i and A_j must have the same outdegree.
- If $u \to v$, then $v \to u$ cannot exist.
- The number of edges is maximum under the above constraints.

EXPLANATION:

Suppose A has K distinct values, $0, 1, 2, \ldots, K-1$.
Let's consider the cases $K > 2$ and $K = 2$ separately.

K > 2

This case is in fact rather trivial.

For each $0 \le i < j \le K-1$, add an edge from every vertex with color i to every vertex with color j.
This clearly satisfies every condition except strong connectivity.

Now, reverse all the edges from color 0 to color $K-1$. This keeps the number of edges the same, but also gives us strong connectivity.

Every pair of vertices with distinct colors has an edge between them in one direction, so this is of course the maximum we can achieve.

K = 2

This case is the meat of the problem.

Suppose there are X vertices labelled 0 and Y vertices labelled 1. Without loss of generality, let $X \ge Y$.

Suppose the respective outdegrees are A and B, for a total of $AX + BY$ edges. Our aim is to maximize this quantity.
Note that the 1-vertices have a total indegree of AX, so one of them must have $\lceil \frac{AX}{Y} \rceil$ incoming edges.

In particular, this means that $B + \lceil \frac{AX}{Y} \rceil \le X$, since we can't have both $u \to v$ and $v \to u$ as edges.
So, if we fix A, we obtain an upper bound on B.

So, let's try every value of A from 1 to $Y-1$ (any more would be pointless).
For each value of A, obtain an upperbound on B. If this upper bound is positive, look at the value of $AX + BY$ you get and take the maximum among them all.

If no valid pair (A, B) exists, the answer is of course -1.
Otherwise, now that we know A and B let's construct a graph.

There are several ways to do this.

One simple way is to do it greedily:

- First, pick Y vertices labelled 0, and connect them to all the Y 1-vertices in an alternating cycle.
- Next, for each vertex labelled 1, choose the A vertices labelled 0 with smallest indegree (and which don't already have an edge to/from it) and connect edges to them
- Then, for each vertex labelled 0, choose the B vertices labelled 1 with smallest indegree (that don't already have an edge to/from this vertex) and connect edges to them.
  - It's always possible to find B such vertices, our initial computation of A and B ensured this.

This gives a valid graph, and we're done.

TIME COMPLEXITY:

$\mathrm{O}(N^2 \log N)$ per testcase.

PREREQUISITES:

Dynamic programming

PROBLEM:

There is a string S, on which Alice and Bob play a game. Alice starts first and then they alternate moves.

- On her move, Alice picks a prefix of S, appends it to her string, and then removes this prefix from S
- On his move, Bob picks a suffix of S, appends its reverse to his string, and then removes this suffix from S

If they play optimally, what is the maximum possible value of the longest common subsequence of Alice's and Bob's strings?

EXPLANATION:

The main observation to be made here is that, no matter how the moves are performed, Alice's final string will be some prefix of S and Bob's string will be the (reversed) remaining suffix.

That is, Alice's string will be $S[1 \dots i]$ and Bob's string will be $rev(S[i+1 \dots N])$ for some $1 \le i \le N$. It's not hard to see that every such pair of strings is achievable.

For convenience, let R denote the reverse of S.
Our task is then to compute the maximum value of $LCS(S[1 \dots i], R[1 \dots j])$ across all (i,j) such that $i+j = N$.

This is a rather standard problem that is solved using dynamic programming; in fact, we can simply use the exact same states as the classical longest common subsequence problem.

Let `dp[i][j]` denote the longest common subsequence of the strings $S[1 \dots i]$ and $R[1 \dots j]$.
Then, there are three possible cases:

- S_i is not a part of the LCS; in which case the answer is `dp[i-1][j]`
- R_j is not a part of the LCS; in which case the answer is `dp[i][j-1]`
- S_i and R_j are both part of the LCS; in which case the answer is `1 + dp[i-1][j-1]`
  - Note that $S_i = R_j$ must hold for this case to be applicable.
- `dp[i][j]` is the maximum of the above three values.

The base cases are `dp[i][0] = 0` and `dp[0][j] = 0`.

Dynamic programming allows us to compute `dp[i][j]` for every $0 \le i,j \le N$ in $\mathrm{O}(N^2)$, after which the final answer is simply the maximum value of `dp[i][j]` across all (i,j) such that $i+j = N$.

TIME COMPLEXITY:

$\mathrm{O}(N^2)$ per testcase.

PROBLEM:

There's an array A, and Q queries on it. You also have an integer X, initially equal to 0.
The i-th query consists of two indices L_i and R_i.
For this query, you add $A_{L_i} + A_{L_i+1} + \ldots + A_{R_i}$ to X.

You're allowed to rearrange A as you like *before* any queries are performed.
What's the maximum possible final value of X?

EXPLANATION:

Suppose A was fixed. Let's look at how X is computed from a different perspective.

For each query (L_i, R_i), let's say we *select* all the indices from L_i to R_i.
Let C_i be the number of times index i is selected across all the queries.
Then, the final value of X is simply $\sum_{i=1}^{N} C_i \cdot A_i$, that is, the sum of A_i multiplied by the number of times index i was selected; across all i.

Notice that the C_i values are completely independent of ordering of A; and depend purely on the query indices.
All the C_i values can be computed in $\mathrm{O}(N + Q)$ time.

How?

Let's start with $C_i = 0$ for all i.
For each query (L_i, R_i), we want to add 1 to each C_j such that $L_i \le j \le R_i$.
Doing this in a bruteforce manner will take $\mathrm{O}(N \cdot Q)$ time.

However, we can do better if we process queries *offline*.
In fact, processing many range-add updates like this offline is a standard trick, and can be done with the help of prefix sums/difference arrays.

The idea is as follows:

- Consider a new array D of length N, where $D_i = C_i - C_{i-1}$ (treat $C_0 = 0$).
  D is the *difference array* of C.
- Note that if we know the values of array D, then finding C is easy: we have $D_1 + D_2 + \ldots + D_i = (C_1 - C_0) + (C_2 - C_1) + \ldots + (C_i - C_{i-1}) = C_i - C_0 = C_i$.
  So, C is simply the prefix sum of the D array!
- Now, look at how queries on C change D.
  If we add 1 to the range from L to R of C, then:
  - D_L increases by 1, because $D_L = C_L - C_{L-1}$ and the latter doesn't change.
  - $D_{L+1}, D_{L+2}, \ldots, D_R$ all don't change, since we're both adding 1 and subtracting 1 from them.
  - D_{R+1} decreases by 1.

So, each update on C only changes two values of D.
Maintaining this is therefore very easy: start with D filled with all zeros; and for each update (L, R), increment D_L by 1 and decrement D_{R+1} by 1.

Finally, obtain C as the prefix sum array of D.

Now, each query takes $\mathrm{O}(1)$ work, and we compute prefix sums once at the end; for $\mathrm{O}(N + Q)$ time in total.

Once the C_i values are known, we want to 'match' the values of A to them so that $\sum C_i \cdot A_i$ is maximized.

This can be done greedily.
That is, match the highest C_i with the highest A_i, the second highest C_i with the second highest A_i, and so on.
The correctness of this follows from the **Rearrangement inequality**.

Implementing this is fairly straightforward: sort the C_i and A_i values in ascending order, then just match them.
To reconstruct the required array, all you need to do is keep a record of the indices of the C_i values after they've been sorted.

TIME COMPLEXITY

$\mathrm{O}(N \log N + Q)$ per test case.

PREREQUISITES:

Familiarity with bitwise operations

PROBLEM:

You have two arrays A and B of length N. An array X is said to be good if it satisfies

$$A_i = X_1 \mid X_2 \mid \ldots \mid X_i$$
$$B_i = X_i \,\&\, X_{i+1} \,\&\, \ldots \,\&\, X_N$$

for *every* i from 1 to N.
Find the maximum value of $X_1 \oplus X_2 \oplus \ldots \oplus X_N$ across all good arrays.

EXPLANATION:

Since everything's a bitwise operation, let's look at each bit separately and check whether we can add this bit to our bitwise XOR.

Consider a fixed bit b. We only need to care about whether b is set or not in each A_i and B_i.
So, we can treat A and B as boolean arrays corresponding to this bit.

Now, let's use the fact that A is prefix OR and B is suffix AND to deduce some properties about which X_i have bit b set.
Since the input guarantees that at least one valid X exists, A and B both look like $[0, 0, 0, \ldots, 0, 1, 1, 1, \ldots, 1]$ (i.e, some zeros followed by some ones). Of course, they might have different numbers of ones and zeros.
Now,

- If $B_i = 1$, then all of $X_i, X_{i+1}, \ldots, X_N$ must have bit b set.
- If $B_i = 0$, then at least one of $X_i, \ldots, X_N$ must have b *un*set.
- In particular, if $B_i = 0$ and $B_{i+1} = 1$, then X_i must have b unset, and $X_{i+1}, \ldots, X_N$ will all have it set. So, X_i is the last time it's unset.
- Similar analysis will tell you that if $A_i = 0$ and $A_{i+1} = 1$, then X_{i+1} must have b set; while $X_1, X_2, \ldots, X_i$ all have it unset. So, X_i is the first time it's set.

Let first_b be the first position i such that X_i has bit b set, and last_b be the last position where it's unset. These can be found as described above.

Let's do a bit of casework:

- If $\text{last}_b + 1 = \text{first}_b$, then we know that exactly some suffix of X (specifically, the one starting from first_b) contains bit b. Depending on the length of this suffix (odd/even), the final XOR will either contain bit b or not.
- If $\text{first}_b + 1 = \text{last}_b$, then the suffix of X starting at $\text{last}_b + 1$ contains bit b, as well as position first_b. Once again, we know exactly how many occurrences there are, so we can tell uniquely whether the answer contains bit b or not.
- Otherwise, $\text{first}_b < \text{last}_b$. In this case, the positions between $\text{first}_b + 1$ and $\text{last}_b - 1$ are 'free', i.e, for each one we can freely decide whether this X_i contains bit b.
  In particular, we can always make the number of occurrences odd, so we can ensure that the answer will always contain bit b.

Processing a single bit takes $\mathrm{O}(N)$ time, so we have a solution in $\mathrm{O}(30 \cdot N)$.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Computing SCCs of a directed graph

PROBLEM:

There are N medicines and M diseases. The i-th medicine increases your strength against disease x_i by 1 and decreases your strength against y_i by 1.
Each medicine can be taken as many times as you like.

How many different subsets of diseases exist such that you can have positive strength against every disease in this subset?

EXPLANATION

Quite often, when you're given a bunch of pairs as input it hints toward an underlying graph.

So, let's construct a graph on M vertices, one corresponding to each disease.
For the i-th medicine (x_i, y_i), add the **directed** edge $x_i \to y_i$ to this graph.

Notice that any *path* (not just edge!) from u to v in this graph allows us to increase $f(u)$ by 1 and decrease $f(v)$ by 1; while keeping all other values unchanged.
Let's try to use this information to figure out what subsets are counted in the answer.

Fix a subset S of medicines; we want to check if we can make $f(u) > 0$ for every $u \in S$.
Let's call a vertex *good* if it's in S, and *bad* otherwise.
It can be seen that:

- If a path $u \to v$ exists, where u is good and v is bad, then we can make $f(u)$ as large as we like by choosing the $u \to v$ path over and over again; after all, we don't care about how low $f(v)$ gets.
- In particular, if every good vertex can reach a bad vertex, we can achieve what we want.
- This leaves us with the case where some $u \in S$ cannot reach any bad vertex.

In fact, if $u \in S$ cannot reach any bad vertex, S is not a valid subset, i.e, we cannot achieve positive strength for every medicine in S.

Proof

Let the set of vertices reachable from u (including u) be $R = \{u_1, u_2, \ldots, u_k\}$.
Suppose they're all good vertices.

Notice that, in particular, every u_i can only reach some subset of R.
Consider the quantity $\alpha = f(u_1) + f(u_2) + \ldots + f(u_k)$. Initially, $\alpha = 0$.
Our aim is to perform operations in such that every $f(u_i) > 0$, which also means $\alpha > 0$ must hold at the end.

But, whenever we choose a path starting at some u_i, it will end at some other u_j.
This both increases and decreases α by 1, making the overall change 0.
If we choose a path that *doesn't* start at a u_i,

- The path can end at some u_i, decreasing α by one; or
- The path can end at some vertex that isn't a u_i, which doesn't change α

This means that no matter how many operations we perform, α can never increase.
In particular, we can never make it positive, which is our goal.
This completes our proof.

We can now categorize all valid S: every vertex in S must be able to reach some vertex outside of S.
This hints to us that we need some reachability information.

So, let's compute the SCCs and resulting **condensation DAG** of our graph.

Consider a 'sink' vertex in the condensation graph, and the SCC it represents.
Notice that S cannot contain *every* vertex in this SCC, since they can only reach each other.
On the other hand, as long as one vertex from this SCC is not in S, we're good: all the others can reach this one vertex.
So, S can contain any subset of this SCC that's not the whole thing, which gives us $2^r - 1$ choices, if r is its size.

Suppose we make this choice for every sink SCC. These choices are independent, so we can multiply their counts all together.
Now, notice that among the remaining vertices, we can pick whatever subset we want!
This is because every non-sink SCC can reach some sink SCC, and the sink has some unchosen vertex so we're good.

In the end, we have a rather straightforward implementation:

- Create the directed graph on medicines as described; and compute its SCCs
- For each SCC, whose size is r,
  - If it's a sink, multiply $2^r - 1$ to the answer.
  - Otherwise, multiply 2^r to the answer.

TIME COMPLEXITY:

$\mathrm{O}(N + M)$ per testcase.

PROBLEM:

You have an array A, and an initially empty array B.
You can perform the following operation:

- Pick a non-empty subsequence of A.
- Append the mex of this subsequence to B.
- Delete the subsequence from A.

Find the lexicographically maximum possible value of B.

EXPLANATION:

Lexicographic minimization/maximization problems often require us to make greedy choices; and this one is no different.

In order to obtain the lexicographically largest array, the first step is to extract the maximum possible mex we can.
That is, find the largest possible integer k such that all of $0, 1, 2, \ldots, k$ exist in the array; giving us a mex of $k + 1$.

It's easy to see that this process can just be repeated:
While the array has at least one element, find the largest possible mex; and delete one instance of every element less than this mex.

We only need to implement this to run quickly enough.
That can be done with the help of a frequency array.

Let $\text{freq}[x]$ be the number of times x occurs in A.
Then, in each step we can do the following:

- Iterate k starting from 0. Stop when $\text{freq}[k] = 0$.
  k is the largest mex we can obtain.
- If $k > 0$, reduce $\text{freq}[x]$ by 1 for every $0 \le x < k$, to simulate deleting the subsequence $\{0, 1, 2, \ldots, k - 1\}$ from the array.
- Otherwise, $k = 0$; meaning that 0 is no longer present in the array.
  This means all further mexes will only be 0.
  So, if there are m elements remaining in the array, append 0 to B, m times; then break out since we've processed everything we can.

The complexity of this is in fact $\text{O}(N)$:

- Building the frequency array takes $\text{O}(N)$ time, obviously.
- Whenever we find a mex of $k > 0$, we iterated $k + 1$ times (and deleted k elements) to do so.
  This means the total sum of non-zero mexes we obtain cannot exceed N; meaning the total number of iterations is also $\text{O}(N)$.
- When $k = 0$, we append a bunch of elements to B and immediately broke out; obviously this is linear time as well.

TIME COMPLEXITY

$\text{O}(N)$ per testcase.

PREREQUISITES:

Combinatorics, specifically **stars and bars**

PROBLEM:

Given N and M, count the number of sequences A of length N such that:

- $0 \le A_i \le M$
- $|A_i - \text{MEX}(A_1, A_2, \ldots, A_i)| \le 1$

EXPLANATION:

Let's first analyze the structure of an absolute mex sequence.

Suppose we've placed $A_1, A_2, \ldots, A_{i-1}$, and want to decide on A_i.
Let $m_{i-1} = \text{MEX}(A_1, \ldots, A_{i-1})$.
There are now two possibilities: we set $A_i = m_{i-1}$, or we don't.

- If we don't set $A_i = m_{i-1}$, then $\text{MEX}(A_1, \ldots, A_{i-1}, A_i) = m_{i-1}$.
  So, we want $|A_i - m_{i-1}| \le 1$ and $A_i \neq m_{i-1}$, which (for the most part) gives us two choices; namely $m_{i-1} - 1$ and $m_{i-1} + 1$.
- If we set $A_i = m_{i-1}$, then $\text{MEX}(A_1, \ldots, A_{i-1}, A_i) > m_{i-1}$. In particular,
  - If $m_{i-1} + 1$ has occurred in the sequence before this, the new MEX is $m_{i-1} + 2$, which is bad for us since $|A_i - (m_{i+1} - 2)| = 2$.
  - If $m_{i-1} + 1$ hasn't occurred yet, the new MEX is $m_{i-1} + 1$, which is fine.

Together, these observations give us a somewhat rigid structure on what an absolute mex sequence looks like.

- Let's say the sequence goes 'bad' at index i if we place $m_{i-1} + 1$ at it.
- Notice that as soon as a sequence goes bad, every following element must be either $m_{i-1} + 1$ or $m_{i-1} - 1$; since as we noted above, placing m_{i-1} would cause the difference to be 2 which is unacceptable.

So, it's enough to count the following:

- The number of sequences that never go bad; let's call them good sequences
- For each i, the number of sequences that go bad for the first time at index i.

These can be done separately.

Counting good sequences

It's not hard to observe that a good sequence will be of the form

$$[0, 0, 0, \ldots, 0, 1, 1, \ldots, 1, 2, 2, \ldots, 2, 3, 3, \ldots, k]$$

for some $0 \le k \le M$.
That is, some zeros, followed by some ones, followed by some twos, and so on.

Counting the number of such sequences is fairly easy.
Let's fix k, the largest element of the good sequence.
Then, we only need to decide how many times each of $0, 1, 2, \ldots, k$ occur in the sequence, since it'll be sorted.
Also, each of them needs to occur at least once.

This is the classical **stars-and-bars** problem in combinatorics, and simply equals $\binom{N-1}{k}$.

So, the number of good sequences is simply

$$\sum_{k=0}^{M} \binom{N-1}{k}$$

which can be computed in $\mathrm{O}(\min(N, M))$.
Counting bad sequences

Now, let's count bad sequences.
If the sequence goes bad at $i = 1$, the only possibility is $A = [1, 1, 1, 1, \ldots, 1]$.

For $i \geq 2$, if the sequence goes bad at index i,

- We're placing $m_{i-1} + 1$ at this index; so $m_{i-1} + 1 \leq M$ must hold, i.e, $m_{i-1} \leq M$.
- In particular, indices $1, 2, \ldots, i-1$ must form a good sequence whose mex is m_{i-1}. From above, we know there are $\binom{i-2}{m_{i-1}-1}$ such sequences.
- Positions $i+1, i+2, \ldots, N$ have two choices each: they can be $m_{i-1} - 1$ or $m_{i-1} + 1$. This is 2^{N-i} possibilities.

So, for a fixed i, by considering all possible values of m_{i-1}, we see that the number of sequences is

$$2^{N-i} \times \left(\binom{i-2}{0} + \binom{i-2}{1} + \ldots + \binom{i-2}{M-2}\right)$$

2^{N-i} is easy to compute, but what about the inner summation?

For that, we can use a small trick.
Suppose we know $\left(\binom{i-2}{0} + \binom{i-2}{1} + \ldots + \binom{i-2}{M-2}\right)$ for some index i.
When we move to $i+1$, we'd like to know $\left(\binom{i-1}{0} + \binom{i-1}{1} + \ldots + \binom{i-1}{M-2}\right)$.

Recall Pascal's identity: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

Applying this to our summation, we have

$$\begin{aligned}
\sum_{j=0}^{M-2} \binom{i-1}{j} &= \sum_{j=0}^{M-2} \left(\binom{i-2}{j-1} + \binom{i-2}{j-2}\right) \\
&= \binom{i-2}{-1} + 2 \times \sum_{j=0}^{M-3} \binom{i-2}{j} + \binom{i-2}{M-2} \\
&= 0 + \left(2 \times \sum_{j=0}^{M-3} \binom{i-2}{j} + \binom{i-2}{M-2} + \binom{i-2}{M-2}\right) - \binom{i-2}{M-2} \\
&= 2 \times \sum_{j=0}^{M-2} \binom{i-2}{j} - \binom{i-2}{M-2}
\end{aligned}$$

Notice that we already have $\sum_{j=0}^{M-2} \binom{i-2}{j}$, so we just need to multiply it by 2 and compute $\binom{i-2}{M-2}$ to subtract, and we're done.
This allows to move from i to $i+1$ in $\mathrm{O}(1)$ time, and so we're done.

Note that depending on your implementation, $M = 0$ and $M = 1$ might be special cases (with answers 1 and $N+1$ respectively).

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Familiarity with mex, the **inclusion-exclusion principle**, basic combinatorics

PROBLEM:

You're given a permutation P of $\{0, 1, 2, \ldots, N-1\}$.
Answer Q queries on it, each one as follows:

- You're given L_1, L_2, M_1, M_2
- Find the number of subarrays of P whose lengths lie in $[L_1, L_2]$ and whose mex lies in $[M_1, M_2]$

EXPLANATION:

First, let's look at what a subarray having a mex of M actually entails.
M is the smallest integer not present in the subarray, which in particular means the subarray should contain $[0, 1, 2, \ldots, M-1]$; and should *not* contain M.

Given that P is a permutation, each value appears exactly once in it. Let pos_i denote the position of element i in P.
Notice that there's a unique 'smallest' subarray that contains all the values $\{0, 1, 2, \ldots, M-1\}$; namely the subarray whose left endpoint is $\min(\text{pos}_0, \text{pos}_1, \ldots, \text{pos}_{M-1})$ and right endpoint is $\max(\text{pos}_0, \text{pos}_1, \ldots, \text{pos}_{M-1})$.
Let this subarray be $[l, r]$.

Notice that $[l, r]$ will be contained in *every* subarray whose mex is $\geq M$; and conversely, any subarray containing $[l, r]$ will have a mex that's $\geq M$.

Now let's move on to answering queries.
Having upper and lower bound restrictions on both length and mex value is a bit too restrictive, so let's first try to solve a simpler version of this problem, keeping only the lower bounds.

That is, given M and L, we'll attempt to count the number of subarrays whose length is $\geq L$ and whose mex is $\geq M$. Let's denote this value by $f(M, L)$.

First, let's get a couple of edge cases out of the way.

- If $L > N$, then obviously $f(M, L) = 0$, since no subarray can have length $> N$.
- Similarly, if $M > N$ once again $f(M, L) = 0$ since no subarray can have mex $> N$.

This leaves us with $L, M \leq N$.

Recall from earlier that we in fact categorized all subarrays whose mex is $\geq M$: it's all subarrays containing $[l, r]$, where $l = \min(\text{pos}_0, \text{pos}_1, \ldots, \text{pos}_{M-1})$ and $r = \max(\text{pos}_0, \text{pos}_1, \ldots, \text{pos}_{M-1})$.
l and r can be computed in $\text{O}(1)$, since they're just prefix minimums/maximums of the pos array.

Now that we know l and r, notice that any valid subarray $[x, y]$ must satisfy $1 \leq x \leq l$ and $r \leq y \leq N$. Our task is to count the number of pairs (x, y) that satisfy this condition, and also $y - x + 1 \geq L$ (since we want length $\geq L$).
This is now a combinatorics problem, and can be solved in $\text{O}(1)$.

How?

Suppose we fix $1 \leq x \leq l$. Let's count the number of valid y.

Note that y must satisfy:

- $y - x + 1 \le L$, i.e, $y \ge x + L - 1$
- $r \le y \le N$

The smallest valid y is thus $y_0 = \max(r, x + L - 1)$, and the number of valid y is $N - y_0 + 1$ (assuming $y_0 \le N$, of course; otherwise the number of valid y is zero).

This gives us a solution in $\mathrm{O}(N)$ by iterating x, but we need to do a bit better to answer queries.
So, let's deal with the $\max(r, x + L - 1)$ cases separately; i.e, treat the case when r is the maximum separately from when $x + L - 1$ is the maximum.

r is the maximum

When $r = \max(r, x + L - 1)$, this means the number of valid y for this x is simply $N - r + 1$, a constant (since we can pick $y = r, r + 1, r + 2, \ldots, N$).
So, we only need to find the number of x that satisfy this.

That's not hard. We have two inequalities:

- $1 \le x \le l$
- $r \ge x + L - 1$, or $x \le r - L + 1$

So, $x_0 = \min(l, r - L + 1)$ is the maximum x for which this holds, and there are $(x_0 + 1)$ valid positions (of course, if $x_0 \le 0$ there are 0 valid positions).

This adds $(N - r + 1) \cdot (\min(l, r - L + 1) + 1)$ to $f(M, L)$.

x+L-1 is the maximum

Let's find x_0 as in the previous case.
Now, we need to deal with $x_0 + 1, x_0 + 2, \ldots, l$

Notice that in this case, if $x_0 + 1$ has k valid y-positions, then $x_0 + 2$ will have $k - 1$ valid positions, $x_0 + 3$ will have $k - 2$ valid positions, and so on till l.

So, we'd like to compute the sum of some consecutive range of integers, which is easy to do in $\mathrm{O}(1)$.
Finding the left and right ends of this range can be done by processing $x_0 + 1$ and l, and finding their respective position counts.
The exact details here are left as an exercise to the reader 🙂

You may also see the code linked below.

Now that we know how to compute $f(M, L)$ in $\mathrm{O}(1)$, how do we solve the original problem?

That's simple, apply inclusion-exclusion!
For the query L_1, L_2, M_1, M_2, the answer is

$$f(M_1, L_1) - f(M_1, L_2 + 1) - f(M_2 + 1, L_1) + f(M_2 + 1, L_2 + 1)$$

each of which are computed in $\mathrm{O}(1)$, so we're done.

TIME COMPLEXITY

$\mathrm{O}(N + Q)$ per test case.

PREREQUISITES:

DFS/DSU, Observation

PROBLEM:

You're given a connected undirected weighted graph with N vertices and M edges.
Answer M **independent** queries on it:

- Add edge (u, v) to the graph with weight 0. Then, compute the weight of the minimum spanning tree of the resulting graph, where the total weight is the bitwise OR of chosen edges.

EXPLANATION:

Let Y be the value of the initial OR-MST weight.
Let's first see how to compute Y.

We'd like to minimize the bitwise OR of the chosen weights, so it's optimal to iterate across bits in decreasing order and see if we can avoid taking the current bit, even at the cost of all lower bits.

That leads us to the following algorithm:

- We'll iterate across bits in descending order from 29 down to 0.
  Let ans be the current answer; initially zero.
  Let E be the current active subset of edges; initially this equals all the edges.
- For a fixed bit b, remove any edge of E that has the b'th bit set. Let E' be the remaining set of edges.
  - if E' is a spanning set of edges for all N vertices, ans needn't have bit b set.
    In this case, replace E with E' and continue on.
  - Otherwise, ans must have the b'th bit set. So, increase ans by 2^b, but continue on to the next iteration without deleting any edges from E.

Checking whether a set of edges spans all vertices can be done in linear time using DFS/BFS; and almost linear time using DSU which.
We run this linear method once for each bit, which is good enough.
At the end of it all, ans is the answer we're looking for.

Now we need to deal with updates.
Of course, running the above algorithm afresh for each new edge would be way too slow.

Instead, let's analyze how a new edge $(u, v, 0)$ would change the answer.
For each bit b from 29 down to 0:

- If b isn't set in ans, then we don't need to use the new edge anyway.
- If b *is* set, then we'd like to check if it can improve our answer.
  - Recall that b is set in the answer only when we couldn't connect the graph without using an edge where b was set; i.e, the edges present divided the graph into > 1 connected components.
  - If there are ≥ 3 connected components, a single edge can't help us anyway so there's no change.
  - If there are exactly two connected components, and u and v are in different components, then this edge does help us reduce the total weight!
    Clearly, when we can do this it's optimal to do so.

This gives us an algorithm:

- Find the *highest* bit b such that the graph at this step of our initial algorithm had exactly two connected components, and u and v are in different components.

- At this stage, we're definitely using the edge $(u, v, 0)$ to connect these components. So, add it to the current edges (recall that we know which edges have been discarded already) and recompute the OR-MST.

Of course, running the entire OR-MST algorithm for each query is still too slow.
However, here is where we'll use the fact that queries are independent.

Notice that if we fix the bit b for which we're using the edge $(u, v, 0)$, the answer is in fact independent of what the actual values of u and v are.
The only thing that matters is that our initial OR-MST algorithm gave us exactly two components, and we compute the OR-MSTs of the union of these two components, along with some 0-weight edge joining them.

So, we only need to run the OR-MST algorithm (at most) once for each bit, making 31 runs in total including the initial one.
The results of these runs can be cached so that each query can be quickly answered later.

Putting everything together, our final solution is as follows:

- Run the OR-MST algorithm described initially.
- For each bit such that the resulting graph had exactly two components, run the OR-MST algorithm on the union of these two components (along with some 0-weight edge joining them) as well and store their answers.
- Then, for each query (u, v):
  - Find the highest bit b such that the graph for this bit has two components; and u and v are in different components.
  - The answer for this query is then simply the precomputed answer for this bit.
  - If no such bit exists, the answer is simply ans, the initial OR-MST value.

This requires us to quickly check if two vertices lie in the same component, but we can precompute components for each vertex corresponding to each bit, so this can be answered in $\mathrm{O}(1)$ per bit as well.

TIME COMPLEXITY

$\mathrm{O}(B^2 \cdot (N + M) + B \cdot Q)$ per test case, where $B = 30$ for this problem.

PROBLEM:

You're given a tree with N vertices. Answer Q queries on it:

- Given a subset S of vertices of the tree, define the *ugliness* of a vertex x of the tree to be $\max(\text{dist}(x, y))$ across all $y \in S$.
  Find the minimum ugliness across all vertices.

EXPLANATION:

Let's try to solve a simpler version of the problem first: what if the subset S consisted of all N vertices?

Then, we want to find a vertex that isn't too far away from all the vertices of the tree.
In particular, if there's a path of length d in the tree, then *any* vertex is at a distance of at least $\lceil \frac{d}{2} \rceil$ away from one of the endpoints of this path.

So, let D be the length of the longest path in the tree (i.e, its diameter).
Then, the answer is at least $\lceil \frac{D}{2} \rceil$.
It's not hard to see that the answer is indeed exactly this: choose the midpoint of the diameter to attain it (if there are two midpoints, choose either one).

This idea can be extended to an arbitrary subset S of vertices.

If there's a path of length d between two vertices of S, once again we see that the answer is at least $\lceil \frac{d}{2} \rceil$.
Yet again, if D is the longest path between two vertices of S, the answer will be exactly $\lceil \frac{D}{2} \rceil$ by choosing the midpoint of this path.

So, our problem turns into the following:
Given a subset S of vertices of the tree, find the longest path whose endpoints lie in S.

To compute this quickly, **recall** one of the methods to compute the diameter of a tree:

- Fix an arbitrary vertex x.
- Find the furthest vertex from x; let it be d_1.
- Find the further vertex from d_1; let it be d_2.
- d_1 and d_2 are then endpoints of a diameter of the tree.

This algorithm adapts itself nicely to our version:

- Fix an arbitrary vertex $x \in S$.
- Find the farthest vertex from x that lies in S; say d_1.
- Find the farthest vertex from d_1 that lies in S; say d_2.
- d_1 and d_2 are the endpoints of one longest path, so D equals the distance between them.

The problem now is performing the second and third step quickly.
Of course, we can run a full BFS/DFS each time to compute distances to all vertices; however that would take $\mathrm{O}(N)$ time and definitely won't be fast enough to answer multiple queries.

Instead, note that with a bit of precomputation, we can calculate the distances between any two nodes of a tree in $\mathrm{O}(\log N)$ or $\mathrm{O}(1)$ as follows:

- Root the tree at some node, say 1. Then, using a BFS/DFS calculate d_u — the distance of node u from 1 — for every node u.
- Then, we have $\text{dist}(x, y) = d_x + d_y - d_l$, where $l = LCA(x, y)$ is the lowest common ancestor of x and y. l can be found in $\mathrm{O}(\log N)$ time using **binary lifting**, for example.

This allows us to transform our algorithm to $\mathrm{O}(|S| \log N)$ per query from $\mathrm{O}(N)$, since in each step we only compute distances from a vertex of S to all the other vertices of S.

Since the sum of $|S|$ across all queries is bounded by $4 \cdot 10^5$, this is good enough.

TIME COMPLEXITY

$\mathrm{O}((N + Q) \log N)$ per test case.

PREREQUISITES:

Greedy algorithms

PROBLEM:

You're given a large integer A in binary representation as a string of length N.
Find two binary strings B and C such that $A = B - C$, and the sum of counts of ones in B and C is minimized.

EXPLANATION:

For convenience, this explanation will treat A, B, C as if they were reversed and 0-indexed, so that A_i corresponds to whether 2^i is set in A or not and so on.
In terms of implementation, this means you can reverse A, follow the solution below, then reverse B and C before printing them.

The problem essentially asks us to write A using the smallest number of powers of two, except we're allowed to both add and subtract them (addition goes into B, subtraction goes into C).
However, there's one small catch: B and C have to be of length N each, which means we can't use *larger* powers.

Let's first solve this without the restriction on length N — in the end, we can see how it comes into play.

If we weren't allowed subtraction, the best we can do is obviously to just use all the bits set in the binary representation of A. So, let's take this as our starting point and see how to reduce it.

The main power of subtraction is that it allows us to reduce a block of ones into two ones.
That is, we have

$$2^x + 2^{x+1} + 2^{x+2} + \ldots + 2^y = 2^{y+1} - 2^x$$

so we can replace $y - x + 1$ additive powers of two with just two of them.
Of course, if $x = y$, this will replace one power of two with two of them, which is bad so we won't do it.

This gives us a natural greedy solution:

- Let $B = A$ and C be filled with zeros initially.
- Iterate from left to right, i.e from lower bits to higher bits; and find a maximum block of ones in B, say $[L, R]$. In particular, this means $B_{L-1} = B_{R+1} = 0$ must hold.
- If $L = R$, ignore it.
- Otherwise, set $B_{R+1} = 1$ and $C_L = 1$, since we're replacing $2^L + \ldots + 2^R$ by $2^{R+1} - 2^L$.

Note that iterating from lower to higher is important here because it allows us to further replace the bits we add to B.
For example, if $A = 110110$, going from low to high we'd get:

- Step 1: $B = 001110, C = 100000$
- Step 2: $B = 000001, C = 101000$
  whereas if we went high to low we'd get
- Step 1: $B = 110001, C = 000100$
- Step 2: $B = 001001, C = 100100$
  which is not optimal.

This almost solves our problem in $\mathrm{O}(N)$: the only thing that needs to be taken care of is the very end.
That is, notice that when replacing a block of ones, we need to use the next higher position.
However, if this block of ones forms a suffix of the string, the next position doesn't exist, so we can't perform this replacement.

It's not hard to see that if a block of ones forms a suffix like this, then we have no choice: this suffix of ones must be set in B and none of them can be set in C.
So, we can simply delete the suffix of ones, solve normally for the remaining string (now without having to worry about edgecases!) and finally add the ones back in.

This way, we've solved the problem in $\mathrm{O}(N)$.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Sorting

PROBLEM:

Given an array A, find any rearrangement D of it such that:

- D is not sorted, in either ascending or descending order
- $\min(|D_i - D_{i+1}|)$ $(1 \le i < N)$ is minimum across all valid D

EXPLANATION:

Let's get a couple of edge cases out of the way first:

- If $N = 2$, the answer is obviously -1 since any rearrangement is sorted in ascending or descending order.
- Similarly, if A contains only a single distinct element, there's only one ordering (which is trivially sorted), so the answer is -1.

Let's now solve the remaining cases, i.e, $N \ge 3$ and A contains at least two distinct elements.
First, note that $\min(|D_i - D_{i+1}|)$ is minimized when D is a sorted array, since that's what keeps adjacent elements as close to each other as possible.

In particular, let's sort the array A and find an index p such that $A_{p+1} - A_p$ is minimum.
Ideally, if we are able to find a non-sorted rearrangement such that these two elements stay adjacent, we'll be done.

There are several ways to do this.
One of them is follows:

- Consider the two arrays $[A_N, A_1, A_2, \ldots, A_{N-1}]$ and $[A_2, A_3, \ldots, A_N, A_1]$
  Note that this is constructed based on the *sorted* array A, i.e, $A_1 \le A_2 \le \ldots \le A_N$
- At least one of these two arrays will definitely be a valid answer, so find the minimum adjacent difference for both, and print the one with the lower value.
  Note that one of these two might end up being sorted (in descending order), so make sure to check for that as well so that you don't print a sorted array.

Why does this work?

Let
$X = [A_N, A_1, A_2, \ldots, A_{N-1}]$
$Y = [A_2, A_3, \ldots, A_N, A_1]$

Since A contains at least two distinct elements, neither X nor Y are sorted in ascending order.

Now, suppose X is sorted in descending order.
This tells us that $A_N \ge A_1 \ge A_2 \ldots \ge A_{N-1}$.
Note that we already know that $A_1 \le \ldots \le A_N$ and A contains at least two distinct elements.
Together, this tells us that $A_1 = A_2 = \ldots = A_{N-1} < A_N$.

It's not hard to see that this makes the array Y unsorted and (when $N \ge 4$) have two adjacent equal elements (its first two elements) which is the best-case scenario.

Similar analysis applies to when Y is sorted in descending order, when you'll have A_1 as the unique minimum and all the other elements equal.

If neither X nor Y are sorted, then it's not hard to see that at least one of them preserves the minimum adjacent difference, since the pair $(p, p+1)$ we found in A can only be 'broken' if:

- $p = 1$, in which case it remains intact in X
- $p + 1 = N$, in which case it remains intact in Y

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per testcase.

PROBLEM:

You have an array A of length N, such that $2 \le A_i \le M$.
In one move, you can select an integer X and set $A_i \leftarrow \gcd(A_i, X)$ for every i.

Find the minimum number of moves needed to make all the array elements equal.

EXPLANATION:

If all the A_i are already equal, the answer is 0. We need to solve for the case when they aren't.

Notice that choosing $X = 2$ sets every A_i to either 1 or 2.
Choosing $X = 3$ after this sets everything to 1, since $\gcd(1, 3) = \gcd(2, 3) = 1$.

So, it's always possible to make everything equal within two moves. All we need to check is whether everything can be made equal in 1.

Suppose we are able to do this, and make all the array elements equal to Y in the end.
There are two possibilities here: either $Y = 1$, or $Y > 1$.

Y = 1

We want to choose $2 \le X \le M$ such that $\gcd(A_i, X) = 1$ for *every* A_i.

In particular, this means that X cannot share a prime factor with any of the A_i; if it did, then the corresponding gcd wouldn't be 1.

So, suppose we have a list of all the prime factors of all the A_i, say L.
We need to check if there's an X between 2 and M that doesn't have a prime factor in L.
Notice that it's enough to restrict ourselves to the case when X is prime.

So, we simply need to check whether there's any prime in the range $[2, M]$ that isn't in L.
If the primes upto M are precomputed (using a sieve), this is easy to do (given that you have L in hand, of course).

Finally, we also need to discuss how to compute the list L quickly, i.e, how to prime-factorize all the A_i quickly.
The constraints are low enough that directly factorizing each number in $\mathrm{O}(\sqrt{A_i})$ is probably fast enough, but there's a faster way using the fact that $A_i \le 5 \cdot 10^5$.

In the initial sieve we do to find prime numbers, also store one prime divisor of each number, let's call it $\mathrm{prm}[x]$ for x.
Then, to prime factorize A_i:

- $\mathrm{prm}[A_i]$ is one prime factor.
- The next prime factor can be found by dividing $\mathrm{prm}[A_i]$ out of A_i and again looking at the prm array, i.e, looking at $\mathrm{prm}\left[\frac{A_i}{\mathrm{prm}[A_i]}\right]$.
- Repeat this while $A_i > 1$.

Each division (at least) halves A_i, so this will take $\log A_i$ steps.
This allows us to obtain all the prime factors of the A_i in $\mathrm{O}(N \log M)$ time.

Putting everything together, the solution for this case looks as follows:

- Quickly prime factorize all the A_i, either using the method outlined above or otherwise.
  Let L be the list of all prime factors of all the A_i.

- Then, for each prime p from 2 to M:
  - If p is present in L, continue on. This can be checked quickly using, say, binary search.
  - If p is *not* present in L, then performing one operation with p is valid, and you can **break out immediately**.

Note that as long as you break out immediately, this is fast enough even though there's no constraint on the sum of M across all tests.
This is because the list L contains at most $N \log M$ elements. So,

- If a valid prime is found, it'll certainly be within $N \log M + 1$ steps at most, each taking one binary search. This is not a problem.
- If no valid prime is found, then $N \log M$ must exceed the number of primes from 1 to M, which is approximately $\frac{M}{\log M}$ by the **prime number theorem**.
  This roughly gives us a lower bound on N, something like $N \geq \frac{M}{\log^2 M}$.
  Since the sum of N across all cases is bounded, we're fine.

Y > 1

Suppose we're able to choose an X such that $\gcd(A_i, X) = Y$ for every i, and $Y > 1$.

Notice that Y must be a factor of both X and A_i — in fact, Y must be a factor of *every* A_i.
This means Y must be factor of $\gcd(A_1, A_2, \ldots, A_N)$, by definition of $\gcd$.

In particular, if $\gcd(A_1, \ldots, A_N) = 1$, then such a Y cannot exist.
On the other hand, if $\gcd(A_1, \ldots, A_N) > 1$, then simply choosing $X = \gcd(A_1, \ldots, A_N)$ will also give us $Y = \gcd(A_1, \ldots, A_N) > 1$, and we'll be done!

Check both cases above to see if using one move is possible.
If both cases fail, then the answer is 2, using $X = 2$ and $X = 3$ as discussed above.

As an aside, the technique of prime-factorizing numbers using a sieve is fairly common, and even appeared in **GCD_QUERIES** from last week.

TIME COMPLEXITY

$O(N \log^2 M)$ per test case with a sieve upto 10^6 as precomputation.

PREREQUISITES:

Frequency tables

PROBLEM:

You have an array A. **Exactly** once, you can choose a subarray $[L, R]$ and increase all its elements by 1.

Suppose the final array is B. Find the maximum value of $inv(A) - inv(B)$.

EXPLANATION:

Let's analyze what choosing subarray $[L, R]$ does for the number of inversions, and whether this tells us anything about what choices of L and/or R can possibly be optimal.

Analysis

Let's denote the prefix $[1, L-1]$ by P, the subarray $[L, R]$ by M, and the suffix $[R+1, N]$ by S.

Now, consider some pair (i, j). Let's do a bit of casework.

- If $i \in M$ and $j \in M$, A_i and A_j both don't change, so this pair's contribution to the number of inversions doesn't change.
- If $i \in M$ and $j \in M$, both values increase by 1 so once again, its contribution to the number of inversions doesn't change.
- If $i \in M$ and $j \in S$, then only A_i increases by 1.
  - If (i, j) was already an inversion (i.e, initially $A_i > A_j$), it continues to remain one.
  - If $A_i = A_j$ initially, this pair creates a new inversion
  - If $A_i < A_j$ initially, this pair continues to not be an inversion
- If $i \in P$ and $j \in M$, then only A_j increases by 1.
  - If $A_i \leq A_j$, this pair continues to not be an inversion.
  - If $A_i > A_j + 1$, this pair continues to be an inversion.
  - If $A_i = A_j + 1$, this pair stops being an inversion.

From this, we see that the only way to *reduce* inversions is between P and M; while interactions between M and S are bad because they can increase the number of inversions.
In particular, it'd be nice if P and M were as large as possible, while S was as small as possible.

This is easy to achieve: simply choose $R = N$ always, so the suffix S will be empty!
However, we can't yet say anything about L.

Now that we've fixed R to always be N, we need to find which value of L is optimal.
Checking each one in $\mathrm{O}(N)$ (or worse) is obviously too slow.

Instead, let's be a bit smarter.
Suppose we (somehow) knew the answer for $[L, N]$ (that is, you know how many inversions it reduces).
Can we then compute the answer for $[L+1, N]$?

Yes we can!

Recall from our previous analysis that the only reductions in inversions come from pairs (i, j) such that $A_i = A_j + 1$.

When moving from L to $L+1$, we're essentially moving the element A_{L+1} from the subarray M to the subarray P. So,

- if P contains x occurrences of $A_{L+1}+1$, these x positions were originally reduced inversions with position $L+1$, but they are no longer reduced. So, decrease the current answer by x.
- On the other hand, if there are y occurrences of $A_{L+1}-1$ in M, these y positions now are reduced inversions with position $L+1$, so increase the current answer by y.

We need to be able to quickly compute x and y. Note that they're both frequencies.

So, maintain two frequency tables: one corresponding to P and one corresponding to M.
When moving from L to $L+1$, update the frequencies appropriately: this takes one operation in each table, after which both x and y can be obtained in $\mathrm{O}(N)$ by just looking at the appropriate frequency table.

This allows us to move from $[L,N]$ to $[L+1,N]$ in $\mathrm{O}(1)$ time; updating the answer along the way.
So, start from $L=1$ (for which computing the answer is trivial), and then increase L till N; each time computing the answer for that suffix.

The final answer is the maximum among everything computed.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Basic bit manipulation

PROBLEM:

You are given an array A, and you'd like to move from position 1 to position N.
From position i, you can move to any position j such that $i \le j \le \min(N, i + A_i)$.

The cost of a path is the bitwise OR of all values you visit along the way.
Find the minimum cost, or say that no valid path exists.

EXPLANATION:

Since $2^0 + 2^1 + \ldots + 2^i < 2^{i+1}$ for every $i \ge 0$, it's better for us to not use higher bits as much as possible; even at the cost of using more lower ones.

Let's build our answer bit-by-bit, going from the higher bits to the lower ones (i.e, from 20 down to 0).
Suppose we're currently dealing with bit b. Then,

- We need to (somehow) check if a path exists where we don't need to use bit b.
- If such a path doesn't exist, add 2^b to our answer; otherwise do nothing. Then, move to bit $b - 1$ and continue.

Now let's focus on checking whether a path that doesn't include bit b exists.

- First, since we're iterating in descending order, we know already for every bit $> b$ whether it will be in the answer or not. If some bit doesn't need to be in the answer, let's call it *forbidden*.
- Bit b itself we don't want to include, so let's treat it as forbidden too.
- Any bit $< b$ is 'free', so let's just assume they're all allowed, i.e, none of them are forbidden.

Now that we have a set of forbidden bits, we cannot land on any value that contains a forbidden bit.
Among the remaining positions, we need to check whether a path from 1 to N exists.

This can be done greedily:

- First, neither A_1 nor A_N can contain a forbidden bit, since we'll definitely take those values. Check that first.
- Let's keep a variable reach, denoting the furthest index we can reach. Initially, $\text{reach} = 1 + A_1$.
- Then, for each i from 2 to N:
  - If $i > \text{reach}$, then it's impossible to reach position i (or any position larger than i), so no path exists.
  - If A_i contains a forbidden bit, ignore this index.
  - Otherwise, set reach to $\max(\text{reach}, i + A_i)$.

If we are able to reach position N using this process, a path exists; otherwise it does not.

This allows us to check a fixed bit in $\mathrm{O}(N)$, thus making the whole algorithm run in $\mathrm{O}(20 \cdot N)$.

TIME COMPLEXITY:

$\mathrm{O}(20 \cdot N)$ per testcase.

PREREQUISITES:

Subset sum dp

PROBLEM:

You're given an array A containing N integers.
In one move, you can pick $1 \le i < j \le N$ and subtract $\min(A_i, A_j)$ from both A_i and A_j.

Find out whether it's possible to make all the array elements 0.
If it is, also find a way to achieve this using at most N operations.

EXPLANATION:

First, let's see how to decide whether the answer is `Yes` or `No`.

Clearly, performing an operation when $\min(A_i, A_j) = 0$ is pointless, so let's not perform such operations.

Suppose a valid sequence of moves exists. For each i, let's match it with the j that sets it to zero.
In other words, let's create a graph on N vertices, where for each move (i, j) that we make, we add the edge $i \leftrightarrow j$.

A bit of observation should tell you that this graph cannot contain cycles, i.e, it's a forest.

Proof

Let's build the graph edge-by-edge.
Suppose we're adding the edge (i, j). This sets at least one of A_i and A_j to zero.

Claim: At any stage, every connected component of our graph contains at most one node with non-zero value.
Proof: The proof is by induction on the number of edges added.
The claim is clearly true for when there are no edges, since each component is a singleton.
Now, look at what happens when we add an edge (i, j).

- If A_i and A_j are in the same component, one of them is zero so we don't make this move anyway.
- If they're in different components, these two components join together. However, at least one of A_i and A_j becomes zero, so the new combined component also has at most one non-zero vertex.

This shows that every edge joins two different components together, and hence cannot create a cycle.

This graph being a forest specifically means that it's **bipartite**.
Let (X, Y) be two sets forming a bipartition of this graph.

Also, let $S_X = \sum_{x \in X} A_x$ and $S_Y = \sum_{y \in Y} A_y$ be the sums of the respective sets.

Notice that any edge connects some element of X to some element of Y; in other words, every operation subtracts the same value from both S_X and S_Y.
So, $S_X = S_Y$ must hold.

In particular, this means A can be partitioned into two subsets of equal sum.
Of course, this condition is necessary for the answer to be `Yes`. Is it also sufficient?

Yes, it is!

Suppose X and Y partition A, such that $S_X = S_Y$.

it's quite easy to construct a sequence of operations: pick some non-zero element from X, pick some non-zero element from Y, then perform the operation on them.

Since their sums are equal, both sets will become empty at the same time.

So, all we really need to do is to check whether A can be partitioned into two subsets with equal sum.
This is, in fact, an instance of the subset-sum problem. We're really just looking for a subset of A whose sum equals $sum(A)/2$.
Notice that if $sum(A)$ is odd then we can never find such a subset.

The subset-sum problem can be solved in $\mathrm{O}(N \cdot sum(A))$ using dynamic programming.

How?

Let $dp(i, x)$ be True if we can get a subset with sum x from the first i elements, and False otherwise.
Then, $dp(i, x)$ is True if and only if $dp(i-1, x)$ is True, or if $dp(i-1, x-A_i)$ is True; otherwise it's False.

This gives us $\mathrm{O}(N \cdot sum(A))$ states and $\mathrm{O}(1)$ transitions each.

Once a bipartition has been found, reconstructing the operations is trivial; as mentioned above, simply keep picking one non-zero element each from the two sets.
This takes at most $N-1$ steps, since every operation makes one element 0.

This instance of the subset sum problem can be solved even faster: using bitsets for a factor 64 speedup, or even in $\mathrm{O}(N \cdot \max(A))$ (see **this blog** if you're interested).
However, neither are needed to get AC in this task.

TIME COMPLEXITY:

$\mathrm{O}(N \cdot sum(A))$ per testcase.

PROBLEM:

You have an array A containing N integers upto 10^7.
Define a sequence of arrays $A^{(k)}$, where:

- $A^{(0)} = A$
- $A^{(k)} = \frac{L}{G} \cdot A^{(k-1)}$ for $k \geq 1$, where G and L are respectively the gcd and lcm of $A^{(k-1)}$

Answer Q queries; each giving you an integer k and asking you to find the gcd and lcm of $A^{(k)}$, modulo $10^9 + 7$.

EXPLANATION:

Let G_k and L_k denote the gcd and lcm of $A^{(k)}$.

Let's first find the answers for $K = 0$, i.e, G_0 and L_0.
Finding G_0 is trivial by just iterating across the array; however, L_0 can be extremely large — computing it directly it impossible.

However, notice that $A_i \leq 10^7$ means that even if L_0 is very large, all its prime factors will be $\leq 10^7$.
This means we can find L_0 in terms of its prime factorization.

Recall that if we have two integers x and y such that

$$x = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$$
$$y = p_1^{b_1} p_2^{b_2} \dots p_k^{b_k}$$

then $\text{lcm}(x, y) = p_1^{\max(a_1,b_1)} p_2^{\max(a_2,b_2)} \dots p_k^{\max(a_k,b_k)}$

Of course, this generalizes to the lcm of more than two integers as well: the power of each prime p_i in the lcm will be the *maximum* power of p_i present in one of the numbers.

Using this idea, let's compute L_0.
Consider some prime $p \leq 10^7$. To know the power of p in L_0, we want to know the maximum power of p across all A_i.

In particular, we have the following algorithm:

- Let mx_p denote the maximum power of p present in some A_i.
  Initially, $\text{mx}_p = 0$ for all p.
- Then, for each i from 1 to N:
  - Find all prime factors of A_i; let them be $p_1, p_2, \dots, p_k$ with respective powers $a_1, a_2, \dots, a_k$.
  - For each $1 \leq i \leq k$, set $\text{mx}_{p_i} \leftarrow \max(\text{mx}_{p_i}, a_i)$

At the end of this process, mx_p will be the value we want, for *every* p.

For this to run quickly, we need to be able to quickly prime-factorize the A_i values.
This can be done using a slight modification to the sieve of Eratosthenes, storing a prime divisor of each integer instead of just a boolean value.
An overview of this can be seen **in this blog** .

Any integer x has $\leq \log_2 x$ prime factors (since each prime factor is at least 2), so the entire process above takes $\text{O}(N \log 10^7)$ time.

Once this is done, L_0 is the product of p^{mx_p} across all primes $\leq p$.

For $k > 0$, notice that for any integers $x, a_1, a_2, \dots, a_k$ we have:

- $\gcd(x \cdot a_1, x \cdot a_2, \dots, x \cdot a_k) = x \cdot \gcd(a_1, a_2, \dots, a_k)$
- $\text{lcm}(x \cdot a_1, x \cdot a_2, \dots, x \cdot a_k) = x \cdot \text{lcm}(a_1, a_2, \dots, a_k)$

That is, multiplying all the numbers of a sequence by a constant multiplies the gcd/lcm of the sequence by the same constant.
Proofs of these can be found **here**.

Further, note that if the gcd and lcm are multiplied by the same constant, their *ratio* remains the same!
In particular, notice that $A^{(k)}$ is obtained from $A^{(k-1)}$ via multiplication by a constant.

This means that $\frac{L_k}{G_k} = \frac{L_0}{G_0}$ for all $k \geq 0$, since each array is obtained in succession via multiplication with a constant.

However, $\frac{L_k}{G_k}$ is also the constant we multiply $A^{(k)}$ with to get $A^{(k+1)}$.
So, we simply have

$$G_k = G_0 \cdot (\frac{L_0}{G_0})^k$$
$$L_k = L_0 \cdot (\frac{L_0}{G_0})^k$$

which can both be found in $\text{O}(\log k)$ time using binary exponentiation, since we already know G_0 and L_0.

TIME COMPLEXITY

$\text{O}(10^7 \log \log 10^7 + N \log 10^7 + Q \log MOD)$ per testcase.

PREREQUISITES:

Math, binary search

PROBLEM:

You're given an array A. In one move, you can replace any element of it with any other integer.

For each i from 1 to N, find the minimum number of moves needed to make the number of modes *exactly* i.

EXPLANATION:

First, note that making the number of modes $> N/2$ but $< N$ is impossible. So, for all such integers, the answer is -1.

Now, let's see how to solve this for a fixed i.
Let's also fix x, the value of the mode itself.

This means that:

- Exactly i integers appear exactly x times in the array. In particular, this means $i \cdot x \leq N$ must hold.
- Every other integer appears at most $x - 1$ times in the array.

This requires us to work with frequencies, so for each integer k, let $\text{freq}[k]$ denote the number of times k appears in the array.
The frequency table can be precomputed with a `map`.

Now, for this fixed i and x, we need to do the following:

- First, bring every element with frequency $> x$, down to frequency x.
  If $\text{freq}[k] > x$, this takes exactly $\text{freq}[k] - x$ moves.
  To find the sum of all of these, it suffices to know the sum of all frequencies that are $> x$, as well as the number of such frequencies. This can be precomputed from the sorted frequency table, and is just a couple of suffix sums.
  Let the number of moves required to do this be s_1.
- Now that everything has frequency $\leq x$, let's look at the number of things with frequency exactly x.
  Suppose there are c such numbers. c can be computed in $\text{O}(1)$ or $\text{O}(\log N)$ (once again, it's a suffix of the sorted frequency array).
- If $c \geq i$, then we need to make $c - i$ of them have frequency $x - 1$, which takes another $c - i$ moves. So, in this case the number of moves needed is $s_1 + c - i$.
- If $c < i$, then we need to bring some elements with frequency $< x$, up to frequency x.
  In particular, we need $x - \text{freq}[k]$ moves for each k with smaller frequency that we pick.

For the last case, it's clearly optimal to choose those k with the largest possible frequency (while also being $< x$).
So, we want the sum of the largest $c - i$ frequencies whose values don't exceed x.

Notice that this represents a subarray of the sorted frequency array, and the right endpoint of this subarray can be computed in $\text{O}(\log N)$ with binary search (it's the largest frequency that's $< x$).
Let this sum be s_2.

Notice that in this case, we need exactly $\max(s_1, s_2)$ moves: when removing 'excess' frequency, we can use the move to increase a lower frequency, and hence no moves are wasted.

So, a fixed i and x can be processed in $\text{O}(\log N)$ or even $\text{O}(1)$ (with two pointers).

As noted earlier, only pairs of i and x such that $i \cdot x \leq N$ matter, and there are $\mathrm{O}(N \log N)$ such pairs (because $\frac{N}{1} + \frac{N}{2} + \frac{N}{3} + \ldots + \frac{N}{N}$ is $\mathrm{O}(N \log N)$).

This solves the problem.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PROBLEM:

You have N cars and M power outlets.
The i-th car has energy capacity A_i Watt-hour.
The i-th output provides B_i Watt power.

If you have H hours, what's the maximum total power that can be charged?
Each car can use at most one charger, each charger can charge at most one car.

EXPLANATION:

Let $K = \min(N, M)$.
Certainly, we can only use at most K outlets, and charge at most K cars.

Since only K outlets can be used, it makes sense to take the best K of them, i.e, the K highest B_i values.
Similarly, it makes sense to take the cars with most capacity - the K largest A_i values.

These can be found by sorting the array in descending order.
Now, let

$$A_1 \geq A_2 \geq \ldots \geq A_K$$
$$B_1 \geq B_2 \geq \ldots \geq B_K$$

We'd like to match each outlet to a car.
It's not hard to see that it's best to match the best outlet to the highest-capacity car, the second best outlet to the second highest-capacity car, and so on.
That is, it's optimal to match A_i with B_i for each $1 \leq i \leq K$. A proof can be found below.

Matching A_i with B_i gives us a total of $\min(A_i, H \cdot B_i)$ energy.
So, after sorting as above, the answer is simply

$$\sum_{i=1}^{K} \min(A_i, H \cdot B_i)$$

Proof of the greedy choice

Suppose A_1 is not matched with B_1.
Let A_1 be matched with B_j, and B_1 be matched with A_i.
Here, $i, j > 1$.

Consider what happens if we match A_1 with B_1 and A_i with B_j instead.

- Initially, A_1 was matched with B_j and B_1 with A_i.
  This provided a total of $\min(A_1, H \cdot B_j) + \min(A_j, H \cdot B_1)$ power.
- After swapping, they'll instead provide $\min(A_1, H \cdot B_1) + \min(A_i, H \cdot B_j)$ power.
- A bit of case analysis should tell you that the second expression is certainly not worse than the first; and is sometimes even larger.

This means it's optimal to match A_1 with B_1.
Once this is done, continue the argument with $i = 2, 3, 4, \ldots, K$ to fix everything.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per testcase.

PROBLEM:

You're given an array A. For each $1 \le i \le N$ independently, find the following:

- If you can change A_i to any **real number**, what's the maximum possible number of inversions in A?

EXPLANATION:

Let $x_1 < x_2 < \ldots < x_k$ be the *distinct* values be present in A.

The main observation required to solve this task is as follows:
Suppose we set $A_i = x$, then there are only about $2k + 1$ 'interesting' values of x, namely:

- One of the x_i; or
- Something less than x_1 (they're all equivalent, so we can treat this as $x_1 - 1$); or
- Something greater than x_k (once again they're all equivalent, so we can consider this to be $x_k + 1$); or
- Something strictly between x_i and x_{i+1} for some i.
  Once again, we can consider this to just be $x_i + 0.5$.

Let's apply this information to solve the problem.

Subtask 1

The easiest subtask allows for pretty much brute-force computation.

As discussed above, there are at worst $2N + 1$ distinct values of x we care about.
So, for each index i:

- Set $A_i = x$ for each of the 'interesting' x.
- Compute the number of inversions in the resulting array.
- Print the maximum across all x.

Even if inversions are computed in $\mathrm{O}(N^2)$ each time, this will run in $\mathrm{O}(N^4)$ time overall which is enough to pass this subtask.

Subtask 2

N is larger now, though still small enough that $\mathrm{O}(N^2)$ or similar can pass.
Earlier, for each i, we set $A_i = x$ and computed the number of inversions in $\mathrm{O}(N^2)$.

Let's optimize the second part.
When we set $A_i = x$, notice that inversion pairs that don't involve index i don't really change.
So, we don't really need to recompute the total inversion number each time.

In particular,

- Let inv be the number of inversions in the original array A.
  Let B_i be the number of inversions involving index i.
- When we set $A_i = x$,
  - Suppose there are y elements before index i that are larger than x.
  - Suppose there are z elements after index i that are smaller than x.
  - Then, the number of inversions in the new array is exactly $\mathrm{inv} - B_i + y + z$.

Of these quantities, inv can be precomputed in $\mathrm{O}(N^2)$, and B_i for all i can be computed at the same time (whenever you see $i < j$ with $A_i > A_j$, add one to both B_i and B_j).
So, if we were able to quickly compute y and z, we'd have a very quick algorithm.

There are several ways to do this, for example:

- Let P_i be a sorted list of the elements $[A_1, A_2, \ldots, A_i]$.
  Similarly, let S_i be a sorted list of $[A_i, A_{i+1}, \ldots, A_N]$.
- Then, when fixing $A_i = x$, y and z can be computed by binary searching on P_{i-1} and S_{i+1} respectively; to find the number of elements in the prefix/suffix that are larger/smaller than x.

Since $N \leq 1000$, computing all the P_i and S_i lists using $\mathrm{O}(N^2)$ memory is not an issue.

We're binary searching $\mathrm{O}(N^2)$ times for a solution in $\mathrm{O}(N^2 \log N)$ overall, which is enough to get AC.
It's possible write the solution to run in $\mathrm{O}(N^2)$ as well (using two-pointers), but that isn't necessary to get AC.
Subtask 3

We can no longer independently try every x for each position.
However, we can still maintain enough information for each one.

Recall that when fixing $A_i = x$, we had the number of inversions be $\mathrm{inv} - B_i + y + z$; where y was the number of elements larger than x before i, and z was the number of elements smaller than x after i.

First off, inv and all the B_i can be computed in $\mathrm{O}(N \log N)$ time: this is a standard application of mergesort or segment/fenwick trees or (in C++) the **order-statistics set**.
A couple of articles detailing this can be found **here** (segment tree approach) and **here** (mergesort approach).

Now, let's define C_x to be the corresponding $y + z$ value for when we set $A_i = x$.
If we knew all the C_x values for index i, we'd just need to take the maximum of them, say M, and the answer would be $\mathrm{inv} - B_i + M$.

Suppose we knew all the C_x values for index i.
Let's see how they change when we move to index $i + 1$.

- First, for any $x > A_{i+1}$, A_{i+1} was contributing a count of 1 to C_x. This will no longer be the case because index $i + 1$ will no longer be to the right.
  So, we need to decrement C_x for all $x > A_{i+1}$ by 1.
- Next, index i will newly become a 'left' index.
  This means, any x such that $x < A_i$ will get 1 added to its C_x value.
- These are the only changes!

Notice that our changes are essentially "subtract 1 from a range of C_x" and "add 1 to a range of C_x"; along with being able to know their maximum after these two operations.
That can be done quickly with a segment tree! (using lazy propagation, of course)

So, at each index we have two range updates and one range query (which is just a query for the whole range).
This can be achieved in $\mathrm{O}(N \log N)$ time, which is good enough for us.

There are a couple of things to take care of:

- First, we need to set the initial C_x values for index 1.
  This is not hard: for each $i > 1$, we must add 1 to C_x for all $x > A_i$; which comes out to $N - 1$ more range updates — our segment tree can handle this.
- More pressingly, we have not-necessarily-integer values of x; as well as very large values of x. Building a segment tree on them is somewhat hard.
  To get around this, we can use coordinate compression.
  Recall that the distinct elements in A were $x_1 < x_2 < \ldots < x_k$.
  Let's relabel them into $1, 3, 5, 7, \ldots, 2k - 1$. This keeps the comparison relation between elements, while also giving us 'space' for real numbers:
  - Anything less than x_1 can be represented by 0
  - Anything more than x_k can be represented by $2k$
  - Anything between x_i and x_{i+1} can be represented by $2i$

After compression, we have an array of size $2N$, and normal segtree stuff can be done on it.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

Greedy algorithms

PROBLEM:

There's an $N \times M$ grid, with $A_{i,j} = i$.

A mountain is a set of integers $P, K, L_1, L_2, \ldots, L_K$, saying that you pick the first L_i integers from row $(P + i - 1)$.

Answer Q queries of the following form:

- Given S, find a mountain with sum S.

EXPLANATION:

This task can be solved greedily.

Suppose we want a sum of S.
Let's go from the first row to the last, each time taking as many numbers as possible till we first exceed the sum.
As soon as we exceed it, we can throw out (at most) one number to attain the exact sum we want.

That is, initialize a variable $\text{sum} = 0$ and set $P = 1$ since we're starting from the first row.
Then, for each i from 1 to N:

- If $\text{sum} + i \cdot M < S$, take all M elements from this row and continue. In other words, set $L_i = M$.
- Otherwise, let j be the smallest integer such that $\text{sum} + i \cdot j \geq S$. Take these j numbers into the sum, i.e, set $L_i = j$.
- Now, if $\text{sum} = S$ we're done.
- Otherwise, remove one element from the row $(\text{sum} - S)$, i.e, decrement $L_{\text{sum}-S}$ by one. Since we took elements in order from smallest to largest, it's guaranteed that the value of $\text{sum} - S$ is no larger than i, so this is always possible.

TIME COMPLEXITY

$\mathrm{O}(N)$ or $\mathrm{O}(N + M)$ per query.

PREREQUISITES:

[Segment trees with lazy propagation]

PROBLEM:

An array B is called good if, when sorted, B_i divides B_{i+1} for every i.

You're given an array A and Q queries on it. For each query (L, R), find the number of good subarrays of $[A_L, A_{L+1}, \ldots, A_R]$.

EXPLANATION:

Let's first analyze the structure of a good array.
Consider a good array B. Without loss of generality, we can assume B is sorted.

Now, for each i, either $B_i = B_{i+1}$ or $B_i < B_{i+1}$.
In the first case, divisibility is trivially satisfied.
In the second case, note that B_{i+1} is at least $2B_i$, since it should be a multiple of B_i that's larger than B_i.

In particular, this means that if B has K distinct elements, the largest of them is at least $2^{K-1} \times B_1$, since each time we move to a higher element the value at least doubles.

In our case, we know $A_i \le N \le 2 \cdot 10^5$.
This means that any good subarray can have at most 18 distinct elements, since $2^{18} = 262144 > 2 \cdot 10^5$.

Solving for $[1, N]$

For now, let's ignore queries entirely and focus on counting the number of good subarrays of a given array.

Let's fix the left endpoint L, and count the number of valid right endpoints R.
Note that if $[L, R]$ is good, then so is $[L, R-1]$: this follows from the fact that if a divides b and b divides c, then a divides c.

In particular, the set of valid R for this L form a contiguous range starting at L, so it suffices to find the right endpoint of this range.

Since the range can only contain upto 18 distinct elements, there's a rather simple way of doing this:
Start with $R = L$.
Then, move R to the next position that contains a new element and check if the resulting set of elements satisfies the divisibility condition.
if it does, jump to the next new element and continue; otherwise $R - 1$ is the position we're looking for.

To do this fast, we need to be able to quickly find the next new element.
This can be done as follows:

- Let S be a set that contains pairs of (position, element), sorted by position. Initially, S is empty.
- Iterate i from N down to 1.
- At i, if A_i exists in S (at some other position), delete it. Then, insert (i, A_i) into S.

S now contains the nearest occurrence of every distinct element that occurs at index $\ge i$, sorted by position. This is exactly what we want.
Now we can simply iterate across S to find the next distinct element, as detailed at the start of the section.

This way, we do N set insertions and deletions, and then at each index upto 18^2 operations (more specifically, upto $\log^2 N$ operations) to find the appropriate R, for a total of $\mathrm{O}(N \log^2 N)$ time.

Answering queries

Of course, applying the above method directly to each query is going to be too slow, since both N and Q are large.

However, notice that we can in fact reuse a lot of information if we solve our queries offline.
That is, first we'll read all the queries, then at a given index i we'll answer all queries whose left endpoint is i.

Consider a new array B of length N. Initially, $B_i = 0$ for every index.

Let's run the algorithm from the previous section. When at a fixed i, first find its optimal right endpoint r_i as before.
Then, add 1 to the range $[i, r_i]$ of B.

Note that B_j now represents the number of indices $\leq j$ (but $\geq i$) such that j is a good right endpoint for this index (since we add 1 to it exactly as many times as it occurs in $[x, r_x]$ for some x).

This means, to answer a query (i, R), we can simply take the sum $B_i + B_{i+1} + \ldots + B_R$.
Note that this is valid only when we haven't yet processed index $i - 1$, which is why we need to solve queries offline.

Maintaining B is theoretically quite simple: we need to support adding 1 on some range, and getting the sum of some range. This is a standard problem that is solved by segment trees with lazy propagation.

TIME COMPLEXITY

$\mathrm{O}(N \log^2 N + Q \log N)$ per test case.

PREREQUISITES:

Sorting, exchange arguments

PROBLEM:

Given the length A_i and beauty B_i of N rods, arrange them in some order to maximize $\sum_{i=1}^{N} x_i B_i$ where x_i is the starting position of rod i.

EXPLANATION:

Let's find the optimal order of rods: computing the answer given this order is easy to do.

First, try to solve the problem for $N = 2$, i.e, you have one rod with parameters (A_1, B_1) and another with parameters (A_2, B_2). How would you determine the optimal order here?

Answer

There are only two possible orders.
The beauties of these two orders are $A_1 B_2$ and $A_2 B_1$ respectively, so the answer is whichever of them is higher.

In particular, notice that rod 1 is placed before rod 2 if and only if $A_1 B_2 \geq A_2 B_1$, i.e, $\frac{A_1}{B_1} \geq \frac{A_2}{B_2}$

A natural generalization of the above observation to N rods would be to sort them in descending order of $\frac{A_i}{B_i}$.
Indeed, this is optimal, and isn't hard to prove.

Proof

Consider some arrangement of rods which is *not* sorted in descending order of $\frac{A_i}{B_i}$.

Then, there must be an index i such that $1 \leq i < N$ and $\frac{A_i}{B_i} < \frac{A_{i+1}}{B_{i+1}}$.
If we swap the places of these two, the beauty increases by $A_{i+1} B_i - A_i B_{i+1} > 0$ so our initial arrangement couldn't have been optimal.

This gives us a pretty simple solution: sort the rods in decreasing order of $\frac{A_i}{B_i}$, then use this order to compute the final answer.

As an aside, what we did above is something called an *exchange argument*, which sometimes can be **quite useful** .

TIME COMPLEXITY:

$\mathrm{O}(N \log N)$ per testcase.

PREREQUISITES:

Greedy algorithms

PROBLEM:

Given N and K, find the smallest integer X such that $N + X$ has at most K distinct digits when written in decimal.

EXPLANATION:

First, if N already has at most K distinct digits the answer is of course 0.

Now, instead of finding X, let's directly find $N + X$; we can subtract N from it in the end.

Let's treat N as a string of digits.
Let the length of this string be L.
Simiiarly, look at $N + X$ as a string of length L (it's obvious that an optimal $N + X$ will also have length L, since $\underbrace{999\ldots99}_{\text{L times}}$ has only one distinct digit and is $\geq N$).

Since $N + X > N$, there must exist an index i such that:

- $N_j = (N + X)_j$ for $1 \leq j < i$
- $N_i < (N + X)_i$

That is, N and $N + X$ will match on some prefix; and when they differ for the first time, $N + X$ will have a higher digit.

Note that if we fix the position at which N and $N + X$ first differ, then all further positions are free: we can place whatever digit we like there, and $N + X > N$ will still hold.
We'll use this to our advantage.

Fix the position i where N and $N + X$ differ.
This fixes the values of $(N + X)_j$ for $1 \leq j < i$.

Now, we want $N_i < (N + X)_i$; but we also want $N + X$ to be as small as possible. A little case analysis tells us the following:

- If $N_i = 9$ then there's no valid choice of $(N + X)_i$, so ignore this i.
- Otherwise, ideally we'd like to choose $(N + X)_i = N_i + 1$.
  - This is almost always possible. The only time it isn't is when it would make $N + X$ have more than K distinct digits, i.e, the set $\{N_1, N_2, N_3, \ldots, N_{i-1}, N_i + 1\}$ has size $> K$.
  - If this happens to be the case, then $(N + X)_i$ should be the smallest element of $\{N_1, N_2, N_3, \ldots, N_{i-1}\}$ that's $> N_i$. If no such element exists, ignore this i; otherwise, it can be found with brute force.
- Now we've fixed $(N + X)_i$, and all the other characters to its right are completely free: we just need to make sure that we still have $\leq K$ distinct digits.
- So, simply choose d to be the lowest digit that makes $N + X$ have at most K distinct digits. Once again, this can be simply bruteforced, since we already know which digits are in the prefix.

This way, we obtain (at most) one value of $N + X$ for every prefix, each of which is valid.

Take the smallest among them as the final value of $N + X$, and thus compute X.

TIME COMPLEXITY

$\mathrm{O}(D \log N \log D)$ or $\mathrm{O}(D^2 \log N)$ per test case, where $D = 10$.

PROBLEM:

Given N, find the lexicographically maximum permutation P of $\{1, 2, \ldots, N\}$ such that $\gcd(P_1, P_3, \ldots) > \gcd(P_2, P_4, \ldots)$; or claim that none exist.

EXPLANATION:

If N is even, the answer is just $[N, N-1, \ldots, 3, 2, 1]$; i.e, the numbers from 1 to N in descending order.
It's easy to see that this satisfies the condition:

- $\gcd(P_1, P_3, P_5, \ldots, P_{N-1}) = \gcd(N, N-2, N-4, \ldots, 4, 2) = 2$, since all these numbers are even.
- $\gcd(P_2, P_4, \ldots, P_N) = 1$ since $P_1 = 1$.
- This is the lexicographically maximum permutation.

If N is odd, the answer is `-1`: no valid permutation exists.

Proof

We want $\gcd(P_1, P_3, \ldots) > \gcd(P_2, P_4, \ldots)$; in particular $\gcd(P_1, P_3, \ldots) > 1$ must be true.
So, there should be some integer $k > 1$ that divides all integers placed at odd positions.

There are $\lceil \frac{N}{2} \rceil$ odd positions, and the numbers placed there need to be distinct.
So, at the very least we need integers $k, 2k, 3k, 4k, \ldots, k \cdot \lceil \frac{N}{2} \rceil$ to be $\leq N$.

However, this is not possible for $k > 1$, and we reach a contradiction.
So, no valid permutation exists.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

Given N, K, S, find a sequence $B = [B_1, \ldots, B_N]$ such that:

- $B_i \in \{-1, 0, 1\}$
- $\sum_{i=1}^{N} B_i \cdot K^{i-1} = S$

EXPLANATION:

Let's try to build B from its first position to its last.

Notice that we can rewrite $\sum_{i=1}^{N} B_i \cdot K^{i-1} = S$ as $S = B_1 + \sum_{i=2}^{N} B_i \cdot K^{i-1} = B_1 + K \cdot x$ for some integer x.

So, $B_1 \equiv S \pmod{K}$.
Notice that this (almost) uniquely determines what B_1 should be:

- If $S \equiv 0 \pmod{K}$, choose $B_1 = 0$
- If $S \equiv 1 \pmod{K}$, choose $B_1 = 1$
- If $S \equiv -1 \pmod{K}$, choose $B_1 = -1$
- If none of the above hold, no S-good sequence can exist.

The only edge case here is when $K = 2$, in which case 1 and -1 are equivalent and you can choose either one.

Now that we know B_1, let's try to find B_2.
A bit of simple algebra tells us that $S = \sum_{i=1}^{N} B_i \cdot K^{i-1}$ is equivalent to $\dfrac{S - B_1}{K} = \sum_{i=2}^{N} B_i \cdot K^{i-2}$.

Notice that the last equation is simply asking us to compute a good array for $\frac{S-B_1}{K}$ of length $N-1$.
So, all we need to do is replace S with $\frac{S-B_1}{K}$ and repeat the above process.

This allows us to compute the values of every B_i.
At the end of the process, if $S \neq 0$ or we couldn't compute some B_i, the answer is -2; otherwise print the B array computed.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

Given N, find three distinct positive integers A, B, C such that the pairwise product of any two is a divisor of N, and the product of all three is a multiple of N.

EXPLANATION:

For convenience, let's suppose $1 \le A < B < C$.
First, note that since AB, BC, AC should all be divisors of N, each of A, B, C must themselves be divisors of N.

We'd like all three to be distinct, so if $N = 1$ or N is prime, the answer is `-1` since N doesn't even have three distinct divisors.

Also note that $C = N$ is impossible.

Why?

We'll always have $B > 1$, so if $C = N$ then BC would be strictly larger than N, and hence cannot be a divisor of it.

This further tells us that if $N = p^2$ for a *prime* p, then the answer is `-1` (since p^2 doesn't have three distinct factors smaller than it).

In every other case, finding an answer is always possible.
One simple construction is as follows: pick $A = 1, B = p, C = N/p$ where p is some prime dividing N. It's easy to verify that this always works when N is neither a prime nor a square of a prime.

So, we need to do the following:

- Check whether N is prime.
- Check whether N is the square of a prime.
- If N is neither, find some prime divisor of N.

This is easy to do in $\mathrm{O}(\sqrt{N})$ by slightly modifying the standard primality check algorithm: simply iterate over numbers from 2 to $\sqrt{N}$, and the first time you find a divisor, return it.
Ensure that the divisor you return isn't the square root of N; and if you don't find any such divisor, the answer is `-1`.

TIME COMPLEXITY:

$\mathrm{O}(\sqrt{N})$ per testcase.

PROBLEM:

You're given an array B of length N, where B_i denotes the parity of the number of occurrences of element i in array A, where A has length N.

Does a valid array A exist?

EXPLANATION:

Let's start with an empty array A and try to build a valid one as we go.

For each i from 1 to N,

- If $B_i = 1$, there are an odd number of occurrences of i in A. In particular, A must contain at least one occurrence of i, so let's add exactly one for now.
- If $B_i = 0$, there are an even number of occurrences of i in A. Let's ignore this i for now.

Our process gives us some array A of length $\leq N$, since it contains at most one occurrence of each i between 1 and N
Notice that we can extend it to length N by adding some more elements, but in order to maintain parity, we can only add an **even** number of any element.

So, if the remaining length is even, the answer is "Yes"; otherwise the answer is "No".

This has a rather simple formulation in terms of the input: the initial length of the array we create is exactly the number of ones in B, which equals $sum(B)$ since B is a boolean array.
So, all that needs to be done is to check whether $sum(B)$ and N have the same parity or not!

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

Chef and Chefina play a game on N piles of stones, the i-th pile having A_i stones. Chef starts and then they alternate moves.
On their turn, a player can:

- Remove one stone from a pile; or
- Remove one stone from every pile
  This can only be done if every pile has at least one stone remaining.

With optimal play, who wins?

EXPLANATION:

When $N = 1$, each player can only remove one stone at a time.
In this case, clearly Chef will win if and only if A_1 is odd.
This should hint at the fact that parity is potentially important.

Each player has two possible moves: removing one stone, or removing N stones.
Let's analyze what happens for different parities of N.

Odd N

When N is odd, every move will remove the same *parity* of stones (since 1 is also odd).

So, if the total number of stones was initially odd,

- After Chef makes a move, it'll be even
- After Chefina makes a move, it'll be odd

In particular, this means Chefina can never win, since an odd number of stones remaining means that there's at least one stone remaining; so she certainly can't be the one making the last move.

Similarly, if the total number of stones was initially even, Chef can never win.

So, when N is odd, Chef wins if the total number of stones is initially odd, and loses otherwise.

Even N

When N is even, removing one stone flips the overall parity, while removing N stones keeps the parity.

As noted in the previous case, if a player can ensure that on their turn there's an odd number of stones remaining, they will always win.
A little analysis should tell you that this is possible when:

- The total number of stones is odd; or
- The smallest pile has odd size

That is, if $S = sum(A_i)$ and $M = \min(A_i)$, then Chef wins if S is odd or M is odd, and loses otherwise.

Proof

We'll show that if *both* S and M are even, Chef cannot win.
There are two possibilities:

- Suppose Chef removes one stone from every pile. Then, since M was even, every pile will still have at least one stone in it.
  So, Chefina can also remove one stone from every pile, and it's back to Chef's turn with both S and M still being even.
- Suppose Chef removes one stone from a single pile. Then,

- If this was from a pile with M stones, it will now have $M-1$ stones, and that's the new minimum. Chefina can also remove one stone from this pile to make the new minimum $M-2$, which is once again even.
- If this was from a pile with $>M$ stones, Chefina is now in a position where the total number of stones is odd but the minimum pile size is still even, being M.

  This means there will still exist a pile with $>M$ stones, and Chefina can remove one stone from it without affecting the minimum.
  Once again, it's Chef's turn with both the number of stones and minimum pile size being even.

So, when both S and M are even, Chefina can always find a move and hence never loses.

Now for the other cases:

- If S is odd and M is odd, Chef can remove one stone from a pile of size M. This makes it Chefina's turn with both S and M now being even, and so she loses.
- If S is odd and M is even, as noted in Chefina's strategy above there will exist a pile with $>M$ stones, so Chef can remove one from it and hence put Chefina in a losing position.
- If S is even and M is odd, Chef can remove one stone from every pile. This reduces S by N (and hence maintains its parity) while reducing M by 1 (hence making it even). Once again, Chefina loses.

So, if either S or M are odd, Chef will win. Otherwise, Chefina wins.

Both cases are easily dealt with in $\mathrm{O}(N)$, since we only need to compute the sum and/or minimum of all the A_i.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

There are two piles of stones: the first with X, the second with Y.
Chef and Chefina alternate moves, with Chef moving first.

- Chef can either remove one stone from both piles or remove two stones from the first pile.
- Chefina can either remove one stone from both piles or remove two stones from the second pile.

Under optimal play, who wins?

EXPLANATION:

Note that Chef always removes at least one stone from the first pile, and Chefina always removes at least one from the second.
So, ideally Chef always wants there to be stones remaining on the first pile, while Chefina always wants there to be stones remaining on the second pile.

Intuitively, this should tell you that both players will use the “remove one stone from both piles” move as much as possible.

This fixes the set of moves of the game, and so it can be easily simulated to see who wins.
A little bit of casework should tell you that:

- If $X \geq Y + 2$, Chef will always win.
- If $X = Y + 1$, Chef will win if X is even and lose otherwise.
- If $X = Y$, Chef will win if X is odd and lose otherwise.
- If $X = Y - 1$, Chef will win if X is odd and lose otherwise.
- If $X \leq Y - 2$, Chef will always lose.

This can be simplified into:

- If $X \geq Y + 2$, Chef wins
- If $X \leq Y - 2$, Chef loses
- if $|X - Y| \leq 1$, Chef wins if and only if $\min(X, Y)$ is odd

In fact, this intuition is correct, and the above cases solve the problem!

Detailed proof

TIME COMPLEXITY

$O(1)$ per test case.

PREREQUISITES:

Tries

PROBLEM:

You have an array A. Process Q independent queries of the following type:

- Given u and x, set $A_u = x$. Then, compute the maximum possible value of

$$((A_1 \oplus \ldots \oplus A_i) \mid (A_{i+1} \oplus \ldots \oplus A_N)) - ((A_1 \oplus \ldots \oplus A_i) \& (\neg(A_{i+1} \oplus \ldots \oplus A_N)))$$

across all $1 \le i < N$.

EXPLANATION:

The given expression is a bit weird, so let's try to simplify it a bit.
For a fixed i, let $x = A_1 \oplus \ldots \oplus A_i$ and $y = A_{i+1} \oplus \ldots \oplus A_N$.

Our expression is $(x \mid y) - (x \& \neg y)$. This simply equals y.

Proof

First, it's clear that $(x \& \neg y)$ is a submask of $x \mid y$ (since it's a submask of x), so the subtraction operation is really just bitwise xor.
Now that we have a bunch of bitwise operations, we can look at bits one by one; in other words it's enough to look at what happens when x and y take the values 0 and 1.

Simple case analysis tells us that:

- $x = 0$ and $y = 0$ evaluates to 0
- $x = 0$ and $y = 1$ evaluates to 1
- $x = 1$ and $y = 0$ evaluates to 0
- $x = 1$ and $y = 1$ evaluates to 1

You'll notice the result is simply y.

So, the answer for a fixed array is the maximum value of $A_{i+1} \oplus \ldots \oplus A_N$ across all $1 \le i < N$, i.e, the maximum suffix xor of the array (except $A_1 \oplus \ldots \oplus A_N$).
That is, if $B_i = A_i \oplus \ldots \oplus A_N$, then the answer is $\max(B_2, B_3, \ldots, B_N)$.

Processing updates

Now we need to process updates on A. Let's see how setting $A_u := x$ changes the answer.
Suppose y is the original value of A_u. Then,

- For any $i > u$, B_i remains unchanged.
- For every $i \le u$, B_i changes to $B_i \oplus x \oplus y$
  - In particular, if we set $k = x \oplus y$, then B_i changes to $B_i \oplus k$.

So, the answer is the maximum of two quantities:

- $\max(B_{u+1}, B_{u+2}, \ldots, B_N)$
  - This is simply a suffix maximum of the B array and can be precomputed.
- $\max(B_2 \oplus k, B_3 \oplus k, \ldots, B_u \oplus k)$

Computing the second part is a rather standard exercise using tries (if you don't know what these are, a tutorial is linked at the start), and can be done in $\mathrm{O}(b)$ where b is the number of bits in the numbers we're dealing with (here, $b = 30$).

However, this requires us to insert $B_2, \ldots, B_u$ into the trie and nothing else, and u changes every query. Continually erasing and reinserting elements would obviously be too expensive.

The simplest way to overcome this is to solve the queries offline.

That is, read all the queries, then we'll process them in increasing order of u.
Let T be a trie, initially empty. Iterate over u from 2 to N.
When at a position u,

- Insert B_u into T
- Then, for each query (u, x), query the trie for the maximum value when xor-ed with $y \bigoplus A_u$.

This gives us a solution in $\mathrm{O}((N + Q) \cdot b)$, where $b = 30$ for this task.

TIME COMPLEXITY:

$\mathrm{O}((N + Q) \cdot b)$ per testcase.

PREREQUISITES:

Familiarity with bitwise operations, (optional) **segment trees**/**sparse tables**

PROBLEM:

You have N integers in a circle. In one move, you can pick two adjacent integers and replace both of them with either their bitwise AND or their bitwise OR.

Find the maximum possible difference between the remaining two elements.

EXPLANATION:

Suppose that x and y are the last remaining elements, with $x \geq y$.
Note that since we merge adjacent elements, x and y are essentially obtained from two disjoint (circular) subarrays of C.

For convenience, let's rotate C so that x is formed from a prefix of C and y from a suffix.
That is, there exists an integer k ($1 \leq k < N$) such that x is formed by a sequence of operations on $C[1:k]$ and y is formed from a sequence of operations on $C[k+1:N]$.

Our objective is to maximize $x - y$. To this end, we have the following observations once k is fixed:

- It's always optimal for x to be the bitwise OR of $C[1:k]$
- It's always optimal for y to be the bitwise AND of $C[k+1:N]$

Proof

Maximizing $x - y$ means making x large and y small. Once k is fixed, these are essentially independent min-max problems, so they can be solved separately.

Clearly, x can be no larger than the bitwise OR of $C[1:k]$, since both operations are bitwise operations.
Similarly, y can be no smaller than the bitwise AND of $C[k+1:N]$.

Both limits are attainable, and hence optimal.

This already gives us a solution in $\mathrm{O}(N^2)$.
We can fix the 'first' element of the array to convert the circle into a line, as above.
Then, we can fix a value of k from 1 to $N-1$; and that gives us x and y as some prefix OR/suffix AND.

We now have to do two things: properly deal with the fact that we're on a circle (and not an array), and speed this solution up.

Converting the circle to an array is quite easy: let's duplicate C and append it to itself, i.e, create the array $[C_1, C_2, \ldots, C_N, C_1, C_2, \ldots, C_N]$.
Notice that in this new $2N$-length array, any subarray of length N corresponds to a rotation of C.
So, converting the circle to a line is simple: fix a value of i, then consider the subarray of length N starting from index i.

For a fixed starting point, our current solution takes $\mathrm{O}(N)$ time (we go through every value of k).
We'll now improve this.

We'll use the fact that once the first element is fixed, there can't be too many different values of x. In fact, there are at most $1 + \log_2 2^{30} = 31$ distinct prefix ORs.

Why?

Let P_i denote the i-th prefix OR of the array.
For any i, either $P_i = P_{i-1}$ or $P_i > P_{i-1}$.

When $P_i > P_{i-1}$, that means C_i has at least one *new* bit, that hasn't appeared in the first i positions.
However, there are only 30 bits in total among the array values. This means that once the prefix OR has increased 30 times, it can no longer increase — all the bits will be set.

This gives us at most 31 distinct prefix ORs in total.

So, instead of fixing a value of k and computing x, let's fix the value of x and compute k and y.
This gives us $\mathrm{O}(30 \cdot N)$ possible pairs of (x, y); so compute them all and take the maximum difference among them.

All that remains is actually computing k and y once x is fixed.

How do I do that?

We want to quickly find the positions where the prefix ORs change.
This can be done by finding, for each bit that's not set in the current prefix OR, its next closest appearance in the array. The minimum of all such values is what we're looking for.

This can be found if we do a bit of precomputation.
Let next be a 2-D array such that $\mathrm{next}(i, b) = j$ means $j \geq i$ is the first position such that A_j has bit b set.
This can be computed by iterating over i in descending order, as:

- If A_i has bit b set, then $\mathrm{next}(i, b) = i$
- Else, $\mathrm{next}(i, b) = \mathrm{next}(i + 1, b)$

With this precomputed, we're good to go!

Let's say that we're considering the rotation starting at i, i.e, the subarray $[i, i + N - 1]$.
Then, the prefix ORs can be computed as follows:

- Let cur denote the current prefix OR, and pos denote the position of the prefix giving that OR. Initially, $\mathrm{cur} = C_i$ and $\mathrm{pos} = i$.
- Finding the next prefix OR is now easy: for each bit b that's not set in cur, look at $\mathrm{next}(\mathrm{pos}, b)$ and take the minimum of all such positions that you find.
- This then tells you the new value of pos.
- After updating pos, cur can be updated as $\mathrm{cur} \mid C_{pos}$.
- Finally, the AND of the remaining array is the AND of everything after pos (upto length N, of course)
  - Notice that this is just a subarray AND query, and can be quickly computed using a segment tree or sparse table.
- Once cur and the subarray AND of the remaining part are known, maximize the answer with their difference.

It's also possible to implement this without any such data structures, using only prefix and suffix ANDs and ORs of C — see the editorialist's implementation for this.

TIME COMPLEXITY:

$\mathrm{O}(N \log^2 M)$ per testcase, where $M = 2^{30}$ here.

PROBLEM:

Given an array A of length N, find the number of its subarrays whose bitwise OR is strictly greater than bitwise XOR.

EXPLANATION:

For conveinence, let $\text{OR}(L, R)$ denote the bitwise OR of the subarray $[A_L, A_{L+1}, \ldots, A_R]$, and $\text{XOR}(L, R)$ denote its bitwise XOR.

For any subarray $[L, R]$, it will always hold that $\text{OR}(L, R) \geq \text{XOR}(L, R)$.
This is because any bit that appears in the subarray will be present in the OR but may or may not be in the XOR.
Of course, any bit that doesn't appear in the subarray at all will be present in neither values.

So, instead of counting the number of subarrays for which $\text{OR}(L, R) > \text{XOR}(L, R)$, we can instead count the number of subarrays for which $\text{OR}(L, R) = \text{XOR}(L, R)$; and subtract this from the total number of subarrays.

To facilitate this, we require one more observation:
Suppose we fix the right endpoint R of the subarray. Consider the set of all bitwise ORs ending at R, i.e, the set $end(R) = \{\text{OR}(L, R) \mid 1 \leq L \leq R\}$
Then, $end(R)$ contains at most 21 elements.

Proof

It should be obvious that $\text{OR}(L, R) \leq \text{OR}(L-1, R)$.
If $\text{OR}(L, R) < \text{OR}(L-1, R)$, that means $\text{OR}(L-1, R)$ contains at least one 'new' bit that wasn't set in $\text{OR}(L, R)$, while still containing all the bits already set in $\text{OR}(L, R)$.

Since we're dealing with 20-bit numbers, this addition of a new bit can happen at most 20 times before there are no more bits to add, and hence there are at most 21 distinct values in the set.

Notice that the above proof in fact told us something a bit more powerful: for each $x \in end(R)$, there's a range of indices $[a_x, b_x]$ such that $\text{OR}(L, R) = x$ **if and only if** $a_x \leq L \leq b_x$.

Actually finding these ranges isn't too hard, although there are both painful and painless ways to implement it.

Implementation details

The simplest way to implement this is probably to do something similar to what's done in point 3 of this **blogpost**, which describes the same idea but for GCD instead.

Let $\text{mn}[i][x]$ denote the *lowest* position j such that $\text{OR}(j, i) = x$.
Suppose we've already computed the values of $\text{mn}[i-1]$.
Then, for each $x \in end(i-1)$, we have

$$\text{mn}[i][x \mid A_i] \leftarrow \min(\text{mn}[i][x \mid A_i], \text{mn}[i-1][x])$$

because the bitwise OR value $x \mid A_i$ till i arises from extending the bitwise OR value x from $i-1$, one step to the right.
Don't forget to set $\text{mn}[i][A_I] = i$ as the first step.

Now, $\text{mn}[i][x]$ gives us the left endpoint a_x of the range corresponding to x.
To find the right endpoint b_x, instead find the left endpoint of the bitwise OR that's just less than x, and move one step to its left.

If a map is used to maintain the $\text{mn}[i][x]$ values, the complexity of this is $\text{O}(21N \log N)$, which is good enough.

There are other ways to implement this too, for example:

- Use binary search along with a structure that allows for range OR queries, such as a segment tree or sparse table
- Maintain some bitwise information to be able to quickly 'jump' to the next higher bitwise OR, for example by precomputing the closest element to the left with each bit set.

Now, suppose we've fixed R, and we know the elements of $end(R)$ and their corresponding ranges.
Let's fix an element $x \in end(R)$ and look at its range $[a_x, b_x]$.
We want to count the number of $a_x \le L \le b_x$ such that $\text{OR}(L, R) = \text{XOR}(L, R) = x$.

However, $\text{XOR}(L, R) = \text{pref}[R] \oplus \text{pref}[L-1]$, where $\text{pref}[i] = \text{XOR}(1, i)$ denotes the prefix XOR array of A.
So, we have

$$\begin{aligned} \text{XOR}(L, R) &= x \\ \text{pref}[R] \oplus \text{pref}[L-1] &= x \\ \text{pref}[L-1] &= x \oplus \text{pref}[R] \end{aligned}$$

Since x and R are fixed, we only need to know the number of $a_x \le L \le b_x$ that satisfy this condition.
But this is easy to do: keep a list of positions corresponding to each prefix XOR, then binary search on the list corresponding to $x \oplus \text{pref}[R]$ to find the number of positions in the range $[a_x - 1, b_x - 1]$.

So, we've solved for a single (R, x) pair in $\text{O}(\log N)$.
As noted earlier, there are at most $21 \cdot N$ such pairs, so this is fast enough for the given constraints.

TIME COMPLEXITY

$\text{O}(B \cdot N \log N)$ per test case, where $B = 21$ here.

PROBLEM:

Given an array A, count the number of pairs (i,j) such that $1 \le i \le j \le N$ and $A_i \oplus A_j$ is a palindrome.
It's known that $A_i < 2^{15}$.

EXPLANATION:

A brute force algorithm would be to check for each (i,j) pair whether their XOR forms a palindrome; of course, this is too slow.

Notice that the number of distinct palindromes we care about is really quite small.
In particular, there are only about $\mathrm{O}(10^{\frac{\log_{10} X}{2}})$ (which is about $\mathrm{O}(\sqrt{X})$) palindromes below X, because fixing one half of the palindrome fixes the other half as well.
In particular, only 427 of the numbers upto 2^{15} are palindromes.

Further, recall that XOR is an *involution*, i.e, $x \oplus y = z$ also means that $x = z \oplus y$.

So, instead of fixing i and j and checking if $A_i \oplus A_j$ is a palindrome, we can fix i and the target palindrome; which then uniquely fixes A_j; after which we only need to count the number of valid j.

That is,

- Let Pal be a list of palindromes upto 2^{15}. This can be precomputed, and has length 427.
- Then, for each i from 1 to N, and each x in Pal,
  - Let $y = A_i \oplus x$.
  - We then want to know the number of indices j such that $A_j = y$.
    This can be found in $\mathrm{O}(1)$ using a frequency table.

Make sure not to overcount pairs of indices, since the above algorithm didn't account for the $i \le j$ condition.
That's not too hard: all pairs except (i, i) will be counted twice, so divide by 2 appropriately.

The complexity is something like $\mathrm{O}(430 \cdot N)$ (more formally, $\mathrm{O}(N \cdot \sqrt{\max A})$), which is good enough to get AC.

Note that the frequency table can be implemented using either an array or a hashmap (`std::unordered_map`, `dict`, `HashMap`) and should run in time without much issue.
Using `std::map` will likely be too slow because of the extra $\log$ factor.

TIME COMPLEXITY

$\mathrm{O}(N \cdot P)$ per test case, where P is the number of palindromes not exceeding 2^{15} (approximately 430).

PROBLEM:

You have an array A.
The score of a partition of A into k subarrays $B_1, B_2, \ldots, B_k$ is

$$\sum_{i=1}^{k} (1 - \max(B_i))$$

Find the maximum score across all partitions of A into subarrays.

EXPLANATION:

Given that we're splitting into subarrays, there's a natural dynamic programming setup.
Let dp_i be the maximum score obtainable by partitioning the first i elements of A, with $dp_0 = 0$ being the base case for the empty subarray.

Then, by considering all possible split points, we have

$$dp_i = 1 + \max_{0 \le j < i} (dp_j - \max(A[j+1 : i]))$$

This is easy to compute in $\mathrm{O}(N^2)$, but we need to optimize it further.

Let's look at the structure of the subarrays we consider.
Suppose we fix an index i. Then, as we move the left endpoint j from i to 1, the value of $\max(A[j : i])$ will only increase — after all, we're only adding elements.

So, we have several distinct maximums to consider, and for each maximum, a *range* of elements where it's the maximum.
Note that for a fixed maximum M, with corresponding range $[L, R]$, we only really care about the maximum dp_j value for $L - 1 \le j \le R - 1$.

Let's keep the following information:

- The current list of maximums, in descending order from left to right.
  Notice that this can be represented using a stack.
- The maximum dp value corresponding to the range of each maximum.
- The set of $(max(dp) - M)$ values across all maximums.
  Let this set be S.

Then, when moving from index $i - 1$ to index i:

- Push A_i onto the stack, as a new maximum.
  The singular dp value corresponding to it is dp_{i-1}.
- Then, while the top element of the stack is not less then the previous element, their ranges can be combined into a single one (with A_i being the maximum value of the new range).
  When combining segments, the new maximum dp value of the segment equals the larger of the previous ones.
- Finally, we keep the set S updated.
  Each time, two elements are removed from the stack and one is added to it.
  This corresponds to deleting two elements from S and inserting one; which can be done quickly using a `multiset`.
- Once the above process is finished, we simply have $dp_i = 1 + \max(S)$.

It's easy to see that $\mathrm{O}(N)$ push/pop operations are done on the stack in total; and each corresponds to one multiset operation for $\mathrm{O}(N \log N)$ in total.

There are also ways to implement this using other data structures, such as a segment tree. A couple of these implementations are linked below; in the author's and tester's code.
The base idea of maintaining a stack of maximums remains the same, however.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per testcase.

PREREQUISITES:

Virtual/auxiliary trees , offline path add updates

PROBLEM:

Given a tree where each vertex has some color, define the value of an edge to be the sum of all the colors it disconnects upon removal.
Find the value of every edge.

EXPLANATION:

Let's consider each color separately and see which edges it affects: if we are able to do this quickly enough for each edge, that will solve the problem.

For convenience, let's root the tree at vertex 1.

Fix a color c, and let the vertices with color c be $x_1, x_2, \ldots, x_k$.
Note that the edges that disconnect c are exactly those edges that lie on the path between some x_i and some x_j, so we'd like to find all such edges quickly.

It's easy to see that the set of these edges themselves forms a tree structure, which is called the auxiliary tree of this set of vertices.
Let T_c denote the auxiliary tree corresponding to color c.

It's possible to build the auxiliary tree of $\{x_1, x_2, \ldots, x_k\}$ using $\mathrm{O}(k)$ LCA computations: the details of this may be found here or here .
This allows us to construct the auxiliary tree of every single color in $\mathrm{O}(N \log N)$ or $\mathrm{O}(N)$ time in total.

Now, consider some virtual tree T_c. We want to add c to every edge that lies in it.
While this is possible using tree decomposition techniques such as HLD combined with a lazy segment tree, the constraints of the problem are rather high and such solutions most likely won't pass in time.

Instead, notice that all of our 'updates' are done before any 'queries' (where each query is the final value of an edge).
This allows us to generalize the trick of processing subarray addition updates offline.

What is this trick?

Suppose we have an array A, and we'd like to support Q updates of the form (L, R, x), each telling us to add x to the subarray $[L, R]$.

To do this, we create a new array B, then add x to B_L and subtract x from B_{R+1}.
Finally, take the prefix sums of B: after this, B_i will denote the value that needs to be added to position i.

Instead of subarrays, we'll use paths from a vertex to its ancestor.
Let's create a new array B, with $B_i = 0$ for each $1 \le i \le N$ initially.
Now, consider some $u \in T_c$. Let $p \in T_c$ be its parent (note that p is the parent of u *in the auxiliary tree*).

To add c to the path between u and p, we add c to B_u and subtract c from B_p.

Now, compute the subtree sums of every vertex using a dfs. Let S_u denote the subtree sum of vertex u.
It's not hard to see that S_u is the answer for the edge connecting it to its parent!

Building the virtual trees takes $\mathrm{O}(N \log N)$ time, and they give us $\mathrm{O}(N)$ updates in total after which we perform a simple DFS.
So, we've solved the problem in $\mathrm{O}(N \log N)$ time in total.

A few other problems that use virtual trees are:

- YATP
- Tree regression
- CF 1725E

TIME COMPLEXITY:

$\mathrm{O}(N \log N)$ per testcase.

PROBLEM:

An array is said to be *good* if no two of its adjacent elements are equal, and it has at most local maximum and one local minimum.

You're given an array A. In one move, you can:

- Choose indices i and j, and set A_i to A_j for a cost of 0.
- Choose index i, and set A_i to any integer for a cost of K.

Find the minimum number of moves to obtain an array that can be rearranged into a good array.

EXPLANATION:

Suppose A is a *good* array. Let's make some observations about it.

Let mx denote the maximum element of A, and mn denote the minimum.
Notice that if either mx or mn appear in the middle of A, they'll be a local maximum/minimum (because adjacent elements can't be equal).
This immediately limits us to at most three occurrences of each one: the endpoints of the array, and one in the middle.

Further, between any two occurrences of mx there *will* be a local minimum (do you see why?); and a similar constraint holds for two occurrences of mn.
This tells us that there can't be ≥ 3 occurrences of mx in the array; since then we'd have two local minimums and that isn't allowed.

So, each of mx and mn can occur at most twice in the array; taking up upto 4 positions.
Also, two of these four positions are the endpoints of the array.
What about the other elements?

A bit of thinking should tell you that any element that isn't mn or mx can occur at most 3 times.

Proof

The above observations drastically limit the number of distinct elements present in the array.
Upto four elements are minimums/maximums; and every other *distinct* element can occur at most thrice.

So, if there are x distinct elements (that aren't the maximum/minimum), we have the inequality $4 + 3x \geq N$, or $x \geq \lceil \frac{N-4}{3} \rceil$.
In total, this means A must have *at least* $2 + \lceil \frac{N-4}{3} \rceil$ distinct elements.

Now, let's relate this to the operations we have.

- The first operation allows us to set A_i to some other A_j.
  This cannot increase the number of distinct elements present, but it can reduce the frequency of one element and increase the frequency of another.
- The second operation allows us to set A_i to any integer.
  This allows us to add a new distinct element to the array.

Recall from above that we have a lower bound on the number of distinct elements required.
So, we *must* perform the second operation till that lower bound is reached.
If we do this y times, the cost incurred is $K \cdot y$.
Finding y is easy: count the number of distinct elements in A, then see how many more are needed to reach $2 + \lceil \frac{N-4}{3} \rceil$.

Once the lower bound on distinct elements is reached, it can be proved that the first operation is enough to rearrange frequencies so that we hit the required bounds (i.e, minimum and maximum appear at most twice,

everything else at most thrice).
The cost here is 0 since the first operation is free.

This solves the problem!

Note that the solution above relies on the fact that $N \geq 4$, since we subtracted 4 from N.
For $N \leq 3$, the answers can be special-cased:

- For $N = 1$ the answer is always 0
- For $N = 2$ and $N = 3$, the answer is K if all the elements of A are equal, and 0 otherwise.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

Given an array A of size N, count the number of its subsequences that are permutations.

EXPLANATION:

Let's count permutations of length $1, 2, 3, \ldots, N$ separately and add them up.

For a permutation of length K, we need to:

- Choose exactly one 1
- Choose exactly one 2
- Choose exactly one 3

  $\vdots$
- Choose exactly one K

There are no further restrictions, since the order of the chosen elements doesn't matter.

In particular, if there are $\text{freq}[i]$ occurrences of element i in the array, then the number of ways to do the above is simply $\text{freq}[1] \times \text{freq}[2] \times \text{freq}[3] \times \ldots \times \text{freq}[K] = \prod_{i=1}^{K} \text{freq}[i]$.

Adding this up for all K, our final answer is thus

$$\sum_{K=1}^{N} \left(\prod_{i=1}^{K} \text{freq}[i]\right)$$

After computing the frequency array, this value is easy to compute in $\mathrm{O}(N^2)$ time.

To optimize it, note that that we're essentially taking prefix *products* of the freq array, and all of those can be computed in $\mathrm{O}(N)$ time total once freq is known, thus solving the problem in $\mathrm{O}(N)$.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

A pet store has N types of animals, the i-th of which has type A_i.

Alice will buy some of these animals and Bob will buy the rest. Is there a way for them to end up with exactly the same multiset of types of animals?

EXPLANATION:

Each type of animal bought by Alice must have a copy that she didn't buy, so that Bob can buy this copy.

In other words, for a valid split to exist, *every* type of animal must be present an even number of times.
It's easy to see that this condition is both necessary and sufficient.

So, all that needs to be done is to check whether each type of animal occurs an even number of times.
One way to do that is as follows:

- Create an array c of length 100, where c_x denotes the number of times x occurs in A.
  - This is easy to do in $\mathrm{O}(N)$, by simply iterating i from 1 to N and incrementing c_{A_i} by 1 each time.
- Then, check whether every c_x is even.

Alternately, you can also sort A and check whether $A_{2i-1} = A_{2i}$ (in 1-indexing) for every i.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

There are three stacks, initially all empty.
For each $1 \le i \le N$ in order, you place i on top of one of the stacks with equal probability.

Given M, find the probability that the sum of the top values of the first two stacks sums to M.
The top value of an empty stack is considered to be 0.

EXPLANATION:

Note that there are 3^N distinct possible final configurations of stacks — each element has 3 choices, independent of all the other elements.
To compute the required probability, we can instead compute the number of configurations that satisfy our required condition, and divide this by 3^N.

Let's start with a slow solution.

Consider some $0 \le x \le N$. Let's find the probability that x is the topmost element of the first stack, and $M - x$ is the topmost element of the second stack.

Of course if $M - x > N$ the probability is 0, so we'll only deal with those x such that $M - x \le N$ (equivalently, $x \ge M - N$).

If $M - x = x$ the probability is again 0, since we only have one of each element.
So, suppose $M - x \neq x$.
In particular, let $L = \min(x, M - x)$ and $R = \max(x, M - x)$, so $L < R$.
Then,

- For elements $1, 2, 3, \ldots, L - 1$, it doesn't really matter which stack they go onto.
  Each of them has 3 options, for 3^{L-1} in total.
  Note that $L = 0$ is a special case; instead of 3^{-1} the factor here is just 1.
- L *must* go to its appropriate stack.
- Elements $L + 1, L + 2, \ldots, R - 1$ can't go onto L's stack, but can go to either of the other two.
  This is 2^{R-L-1} options.
- R *must* go to its appropriate stack.
- Elements $R + 1, R + 2, \ldots, N$ can only go to the third stack.

So, there are $3^{L-1} \cdot 2^{R-L-1}$ options in total.
This is easy to compute in $\mathrm{O}(\log N)$ using binary exponentiation.

Since the total number of valid configurations is obtained by summing this across all valid x, we have a (slow) solution in $\mathrm{O}(N \log N)$.

Now, let's attempt to optimize this.
Suppose $0 < x < M - x$. Then,

$$3^{x-1} \cdot 2^{M-x-x-1} = \frac{2^{M-1}}{3} \cdot 3^x \cdot 2^{-2x} = \frac{2^{M-1}}{3} \cdot \left(\frac{3}{4}\right)^x$$

Here, $\frac{2^{M-1}}{3}$ is a constant, so we only really need to compute the sum of $\left(\frac{3}{4}\right)^x$ for all valid x.
In particular, the constraints on x are:

- $x > 0$

- $x \geq M - N$
- $x < M - x$, meaning $2x < M$

This gives us lower and upper bounds on the possible values of x, and anything between these bounds is valid.

Computing the sum of $(\frac{3}{4})^x$ for a contiguous range of x means we really just want the sum of a geometric progression!
This can be computed quickly using a formula, as seen **here**.
In particular,

$$a + ar + ar^2 + \ldots + ar^{k-1} = a \cdot (\frac{r^k - 1}{r - 1})$$

This allows us to solve the $x < M - x$ case in $\mathrm{O}(\log MOD)$ time (or even $\mathrm{O}(1)$ if you precompute appropriate powers).
Note that $x = 0$ (i.e the first stack being empty) is also a valid case, and the number of possibilities for it is 2^{M-1} (assuming $M \leq N$, of course). This can be handled separately.

The $x > M - x$ case is symmetric since swapping x and $M - x$ keeps the L and R values the same, which means the answer can just be multiplied by 2.

As noted at the start, the final answer is obtained by dividing the value we have, by 3^N.

TIME COMPLEXITY

$\mathrm{O}(\log MOD)$ per testcase.

PROBLEM:

Chef has an integer X, that's initially 0. He can perform **at most** Y moves, each one as follows:

- Increase X by 2; or
- Decrease X by 1

How many distinct final values of X are possible?

EXPLANATION:

If $Y = 0$, no moves can be performed so the answer is just 1. Now let's analyze $Y > 0$.

Since we're allowed **at most** Y moves, we have some degree of freedom to stop when we want to.
Analyzing positive and negative numbers separately (since we can always have $X = 0$ by doing nothing):

- We can use k moves of the second type to reach $-k$.
  This allows us to reach every number in the range $[-Y, -1]$.
  It's clear that no negative number that's $< -Y$ can be reached, since we have no other way of reducing X.
- We can use k moves of the first type to reach $2k$.
  This allows us to reach every *even* number in the range $[2, 2Y]$; and again clearly we can't reach anything beyond $2Y$.
- As for odd positive numbers, note that if we reach $2k$ and have (at least) one move left, we can then reach $2k - 1$ by using the second move and subtracting 1.
  This allows us to reach every odd number in the range $[1, 2Y - 3]$ for sure; but we can't reach $2Y - 1$ because we have no moves left after reaching $2Y$.

Putting everything together, we can reach:

- 0
- $[-Y, -1]$: there are Y integers here
- Even numbers from 2 to $2Y$: another Y integers
- Odd numbers from 1 to $2Y - 3$: there are $Y - 1$ such integers

Note that the last two combined are just all the numbers in the range $[1, 2Y]$, except for $2Y - 1$.

The total count is just $1 + Y + Y + Y - 1 = 3Y$.
So,

- If $Y = 0$, the answer is 1.
- Otherwise, the answer is $3Y$.

TIME COMPLEXITY

$\mathrm{O}(1)$ per test case.

PROBLEM:

You're given an array B of length N consisting of 1's and -1's.
Consider a new array C, such that $C_i = 2^{i-1} \cdot B_i$ for each $1 \le i \le N$.

Suppose P subarrays of C have a strictly positive sum and Q of them have a strictly negative sum.
Find $|P - Q|$.

EXPLANATION:

Consider a subarray $[L, R]$. A little observation should tell you that:

- If $B_R = 1$, then $C_L + C_{L+1} + \ldots + C_R > 0$
- If $B_R = -1$, then $C_L + C_{L+1} + \ldots + C_R < 0$

Proof

Suppose $B_R = 1$, so $C_R = 2^{R-1}$

Recall that the powers of two satisfy $2^0 + 2^1 + \ldots + 2^{R-2} < 2^{R-1}$, which (by removing a few terms) also means $2^{L-1} + \ldots + 2^{R-2} < 2^{R-1}$.
This can be rearranged to $2^{R-1} - 2^{R-2} - 2^{R-3} - \ldots - 2^{L-1} > 0$, and the LHS of this equation is the minimum possible sum of $C_L + C_{L+1} + \ldots + C_R$ (attained when $B_L = B_{L+1} = \ldots = B_{R-1} = -1$)

The minimum possible sum is positive, so clearly any sum is also going to be positive.

A similar proof holds for the $B_R = -1$ case.

This makes the problem quite simple:

- If $B_i = 1$, then all the subarrays ending at i have positive sum. There are exactly i such subarrays (in 1-indexing).
- Similarly, if $B_i = -1$, then all i subarrays ending at this index have a negative sum.

This allows us to compute P and Q both in $\mathrm{O}(N)$ time (if $B_i = 1$, add i to P; otherwise add i to Q), after which $|P - Q|$ is easily found.

You may also note that, if we let $d = P - Q$, then each index i adds exactly $B_i \cdot i$ to d.
So, a simple way to write down the answer is $\left|\sum_{i=1}^{N} i \cdot B_i\right|$.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

DFS, binary search

PROBLEM:

You have a tree on N vertices rooted at vertex 1, the i-th vertex has value A_i.
The *power* of a vertex is the maximum value present in its subtree.

In one operation, you can pick i and increase A_i by 1.
Find the minimum number of operations required to make the sum of powers of all the vertices at least X.

EXPLANATION

Let p_i denote the parent of node i and B_i denote the power of node i.
Computing the initial values of B_i can be done with a single dfs.
Notice that $B_{p_i} \geq B_i$ for every $i > 1$.

Suppose we decide that we're going to increase the value of node u.
It can only affect the powers of its ancestors.

In particular, if the ancestors of u are (in order from the root) $1 = v_1, v_2, \ldots, v_k = u$, then the powers of some *suffix* of these ancestors will increase, while the rest will remain the same.
More specifically, suppose we increase A_u by m. Then,

- Let i be the smallest index such that $B_{v_i} < A_u + m$.
  - Since the B_{v_i} are sorted (in decreasing order), this i can be found in $\mathrm{O}(\log N)$ with binary search.
- Then, $B_{v_i}, B_{v_{i+1}}, \ldots, B_{v_k}$ will all be set to $A_u + m$, while the other vertices' powers will remain unchanged.
- The increase in the sum of powers can be computed in $\mathrm{O}(1)$ if we had prefix sums of the B_{v_i}.

This analysis gives us one important observation required to solve this problem: the fact that it's enough to perform all our operations on a single node.

Proof

Consider a solution where we increase both A_u and A_v.
Suppose their current values are $C_u = A_u + r$ and $C_v = A_v + s$.

Now,

- Let x be the increase in the sum of powers if we do $C_u \to C_u + 1$
- Let y be the increase in the sum of powers if we do $C_v \to C_v + 1$.

Without loss of generality, let $x \geq y$.
Notice that the s moves done on v so far increased the sum of powers by at most $s \cdot y$ each; because each move affected the power of at most y vertices.

Let's undo these s moves and perform s moves on u instead.
Each such move will increase the power by at least x each; giving us a total increase of at least $x \cdot s$.
Since $x \geq y$, we have at least the same sum of powers, using exactly the same number of moves.

Repeating this process over and over again allows us to reach a solution where only one vertex has its value increased, as claimed.

Now that we know only one vertex needs its value to be increased, let's see if we can figure out how much to increase it by.
Clearly, the larger m is the more the increase in sums of powers is going to be.

If we fix m and u, as noted above the increase can be computed in $\mathrm{O}(\log N)$ (if we had appropriate prefix sums present).
So, for a fixed u, we can find the smallest possible m that allows the sum of powers to exceed X by binary searching on m.

This allows us to solve for a single vertex in $\mathrm{O}(\log N \log X)$, provided we have a list of its ancestors and the corresponding prefix sums.

Creating this list for each vertex would take $\mathrm{O}(N)$ time, which is too slow.
Instead, we can be a bit more clever.

Let's maintain two stacks: the list of ancestors from the root, and the corresponding prefix sums.
Then, start a DFS from vertex 1, doing the following:

- When you enter vertex u, push u onto the stack. When doing this, also push the appropriate prefix sum onto the stack (that is, B_u plus the previous top of the stack).
- Now the two stacks contain exactly the lists we need to solve for u, so do that in $\mathrm{O}(\log N \log X)$ as described above.
- Then, DFS into all the children of u, thereby solving for those as well.
- Finally, when leaving u, pop both stacks once.

This procedure ensures that whenever we're at a vertex u, the two stacks contain exactly the information we need.
We do a total of N pushes and pops each, so this part is $\mathrm{O}(N)$.

The algorithm as a whole thus takes $\mathrm{O}(N \log N \log X)$ time.

TIME COMPLEXITY:

$\mathrm{O}(N \log N \log X)$ per testcase.

PROBLEM:

You have N strings, all initially with value 0. Process Q queries/updates:

- Given (i, k, x), add x to the value of all strings whose length-k prefix equals $S_i[1:k]$.
- Given (i, k, T), add a new string to the list that equals $S_i[1:k] + T$ with value 0.
- Given i, print the value of the i-th string.

EXPLANATION:

Let's tackle a slightly easier version of the problem first, without the second query type - i.e, without adding new strings.

Given that we're dealing with strings, and in particular prefixes of strings, the obvious data structure that comes to mind is a trie.
So, let's put all the S_i into a trie.

Now, for the queries:

- Type 1: given (i, k, x), add x to all strings whose length-k prefix equals $S_i[1:k]$.
  Notice that, if we find the vertex u in the trie corresponding to the position of the k-th element of S_i, all valid strings are exactly those that lie in the *subtree* of u.
  So, we'd like to add x to the values of all vertices in this subtree.
  Finding u quickly is not very hard; you can even save that information when you create the trie and just look it up when needed.
- Type 3: given i, print the value of the i-th string.
  Note that the previous query type had us perform several subtree updates; so all we need to do is to quickly get the current value of the endpoint of this string.

After building the trie, our problem has really turned into a problem on a tree: we'd like to add values to certain subtrees, and get the value of certain vertices.
This is a classical problem.

How do I solve it?

Perform an **Euler tour** of the tree, which "flattens" subtrees into subarrays.

Now, subtree updates and vertex queries instead correspond to subarray updates and point queries on an array - which is a well-known problem that can be solved in several ways.
For example, you can use a segment tree with lazy propagation; or even just a normal segment tree/BIT built on the prefix sums of the array instead.

Now, let's try to incorporate the second query type as well.
Once again, let's construct a trie of all the S_i.
The type 2 query (i, k, T) then corresponds to the following:

- Find the position of the k-th character of S_i in the trie.
- Then, insert T into the trie, starting from this position.

The second part is easy once we've done the first, so let's focus on the first instead.
Suppose the length of S_i is L.
Then, the k-th character of S_i from the front is the $(L-k+1)$-th character of S_i from the back.
Moving backwards in the trie is easy; it corresponds to just moving to the parent.
So, if know the vertex u which is the endpoint of string S_i, we're really just looking for the $(L-k+1)$-th ancestor of u, which can be found quickly using binary lifting.

However, notice that this process cannot really be done online, since it would change the structure of the trie (and hence the Euler tour will change).

Instead, we can process all insertions *offline*.

That is,

- Read all the queries initially, but don't do anything with them just yet.
- First, process all type 2 queries only, to build a single large trie.
- Build the Euler tour of this trie; and a segment tree on the tour.
- Next, go over the queries once again, but this time:
  - Type 1 queries correspond to subtree additions, as before.
  - Type 3 queries correspond to point queries, as before.
  - Type 2 queries however, correspond to point-set updates.
    Notice that since we're using the single megatrie, some values might've been added to vertices even though they technically don't exist yet.
    To counter this, you can reset the values of such elements to 0, which can still be done using the segment tree.

TIME COMPLEXITY

$\mathrm{O}(S \cdot (26 + \log S))$ per testcase, where S is the sum of lengths of all input strings.

PROBLEM:

Given a binary string S, in one move you can flip (i.e, convert $0 \to 1$ and $1 \to 0$) some prefix of S.

Find the minimum number of flips to make S contain a substring of K ones.

EXPLANATION:

Let's fix a substring of length K and see how many moves we need to make it contain only ones.

Suppose the substring is $A = A_L A_{L+1} \dots A_R$. Then,

- If $A_i = 0$, we need to flip a prefix that includes i an *odd* number of times.
- If $A_i = 1$, we need to flip a prefix that includes i an *even* number of times.

This should give you a greedy solution:

- If $A_R = 1$, then ignore it. Otherwise, flip the prefix $[1, R]$.
- Then, repeat this with $A_{R-1}, A_{R-2}, \dots, A_L$ in order.

Analyzing this process, you'll note that the number of flips is in fact determined exactly by the number of positions $L \le i < R$ such that $A_i \neq A_{i+1}$; in particular, if there are x such positions, then:

- If $A_R = 0$ then we need $x + 1$ flips.
- If $A_R = 1$ then we need x flips.

Intuitively, we're breaking the substring into contiguous 'blocks' of zeros and ones; after which each flip turns exactly one block into a block of ones.

This allows us to solve a single substring in $\mathrm{O}(K)$. Repeating this for every substring will take $\mathrm{O}(NK)$, which is too slow.

To improve this solution, notice that the answer for a substring is determined by exactly two things: the value of x (i.e, the number of adjacent pairs of unequal characters), and what the last character is.

So, suppose we know x for the substring starting at L. We can easily obtain it for the substring starting at $L + 1$, since the only pairs that need to be looked at are $(L, L + 1)$ and $(R, R + 1)$ (here, $R = L + K - 1$).

This allows us to move from one substring to the next in $\mathrm{O}(1)$, giving us a solution in $\mathrm{O}(N)$ overall.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Combinatorics, André's reflection principle

PROBLEM:

You're given N distinct integers between 1 and $2N$ as an array P.

Consider all permutations Q of $[2N]$ such that $P_i = Q_i$ for $1 \le i \le N$.
Find the number of distinct prefix maximum arrays among all such Q.

EXPLANATION:

tl;dr

The answer is

$$\binom{3N-M}{N} - \binom{3N-M}{N+1}$$

where $M = \max(P_1, P_2, \ldots, P_N)$.

Most of the difficulty of this problem comes from modelling it correctly.
For now, let's ignore the prefix of length N we're given, and just look at prefix maximum arrays as a whole.

Since the prefix maximum array is always going to be sorted, two such arrays will differ if and only if the *counts* of the elements in them differ.
So, it's enough for us to consider counts of elements in the prefix maximum array.

Let c_i denote the number of times i occurs in the prefix maximum array. Then, we have the following constraints:

- $c_i \ge 0$ for every i
- $c_1 + c_2 + \ldots + c_i \le i$ for every i, because the elements $\{1, 2, \ldots, i\}$ together cannot be prefix maximums more than i times.
- $c_1 + c_2 + \ldots + c_{2N} = 2N$.

Any assignment of integers to c_i that satisfies these conditions will give us a unique prefix maximum array, and it's not hard to see that there will always exist a permutation that achieves such an array.

These contraints can be represented as lattice paths!
In particular, we can do the following:

- Start at $(0, 0)$.
- For each $0 \le i < 2N$, make one rightward move followed by c_{i+1} upward moves.

So, after i rightward moves, we would've made exactly $c_1 + c_2 + \ldots + c_i$ upward moves.

Then,

- $c_1 + c_2 + \ldots + c_{2N} = 2N$ ensures that we will always end up at position $(2N, 2N)$.
- $c_1 + c_2 + \ldots + c_i \le i$ means we will always stay below the line $x = y$ (though we can touch it).

It's not hard to see that any path from $(0, 0)$ to $(2N, 2N)$ that always stays below the line $x = y$ similarly allows us to assign values to each c_i, so all we need to do is count the number of such paths!

Counting the number of right-up lattice paths that don't cross the line $x = y$ is a rather well-known problem, and can be done using the reflection principle to subtract 'bad' paths from the total number of paths.

In fact, when starting from $(0, 0)$ and ending at $(2N, 2N)$, the answer is exactly the $2N$-th Catalan number!
A sketch of the proof for this case can be found **here**.

Let's return to the original problem.
We are given the first N elements of the permutation, which doesn't actually change much: in terms of our path, it simply fixes a prefix of our path.

In particular, if $M = \max(P_1, P_2, \ldots, P_N)$, then the path so far has us end at position (M, N).
So, all we need to do is count the number of paths from (M, N) to $(2N, 2N)$ that stay below the line $x = y$.

By the reflection principle, that works out as follows:

- The total number of paths from (M, N) to $(2N, 2N)$ is $\binom{3N-M}{N}$.
- The 'bad' paths from (M, N) are in bijection with the total number of paths from $(N-1, M+1)$, thus being equal to $\binom{3N-M}{N+1}$

This makes the answer simply

$$\binom{3N-M}{N} - \binom{3N-M}{N+1}$$

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

You have a binary string S of length N. You can delete at most one substring of S.
If this move is performed optimally, find the longest prefix containing all 1's.

EXPLANATION:

Suppose we delete substring $[L, R]$. Let's analyze what the longest prefix consisting of ones looks like.
For convenience, a prefix containing only ones will be called *good*.

There are two cases to consider:

- First, suppose substring $[1, L-1]$ contains a zero. Then, the longest good prefix is simply the longest good prefix of $[1, L-1]$; which also equals the longest good prefix of S.
  Notice that this is completely independent of R.
- Second, it might be the case that $[1, L-1]$ contains only ones. Then, the length of the longest good prefix is $L-1$, plus the longest good prefix of $[R+1, N]$.

The first case is quite easy: the longest good prefix of S can be computed in $\mathrm{O}(N)$, so let's deal with the second case.

We require that $[1, L-1]$ contains only ones; meaning $[1, L-1]$ is a good prefix.
Obviously, it's thus best to choose $[1, L-1]$ to be the longest good prefix possible; which we've already computed above.
This fixes L.

Now, what about the choice of R? Of course, it'd be nice if $[R+1, N]$ had a large good prefix.
We can simply find the longest good prefix for every choice of $R \geq L$, as follows:

- Let's look at R from N down to L. Let $\mathrm{good}(R)$ denote the longest good prefix of $[R, N]$.
- If $S_R = 0$, then $\mathrm{good}(R) = 0$.
- Otherwise, $\mathrm{good}(R) = 1 + \mathrm{good}(R+1)$.

Compute $\mathrm{good}(R)$ this way for every $R \geq L$, and take the maximum value of them.

You may also notice that this is equivalent to finding the longest block of ones in S that isn't a prefix.
Either way, the time complexity is $\mathrm{O}(N)$.

TIME COMPLEXITY:

$\mathrm{O}(1)$ per testcase.

PROBLEM:

You have two binary strings A and B of length N.
In one operation you can reverse any substring of A of prime length.

Is it possible to make A equal B?

EXPLANATION:

Clearly, a necessary condition to make A equal B in the end is for them to have equal numbers of 0's and 1's.

It turns out that this condition is also sufficient!
That is, it's possible to make A equal B if and only if they have an equal number of zeros and ones.

Proof

Note that 2 is a prime, which means we're allowed to swap adjacent elements.

This essentially allows to rearrange A however we like, so if A and B have the same number of zeros and ones we can make them equal; otherwise we can't.

Checking this condition is easy: for example, you can iterate over A and B to compute the required counts in $\mathrm{O}(N)$, or sort A and B and check whether they are equal.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

The score of a binary array equals the number of adjacent pairs of unequal elements in it.

Given a binary array A of length N, find a binary array C also of length N such that $C \neq A$, but $score(A) = score(C)$.

EXPLANATION:

The simplest way to achieve $score(A) = score(C)$ is to ensure that:

- if $A_i = A_{i+1}$, then $C_i = C_{i+1}$
- If $A_i \neq A_{i+1}$, then $C_i \neq C_{i+1}$

Then, any index which contributes to A's score also contributes to C's score and vice versa, so they'll surely have the same score.
However, we also need to ensure that $A \neq C$.

To achieve this, one easy construction is to just invert A.
That is,

- If $A_i = 0$, set $C_i = 1$
- If $A_i = 1$, set $C_i = 0$

This guarantees $A \neq C$ (in fact, none of their indices match!), and it's not hard to see that the adjacency relation is preserved ($A_i = A_{i+1}$ if and only if $C_i = C_{i+1}$) hence their scores are equal.
This solves the problem.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

You're given an array A.
The **rank** of A_i is defined to be $L+1$, where L is the number of elements in the array strictly larger than A_i.

Answer Q queries on this array:

- Given K and R, find the smallest integer x such that you can delete $\leq x$ elements from $A[1:K-1]$, another $\leq x$ elements from $A[K+1:N]$, and have the rank of A_K in the resulting array be $\leq R$.

EXPLANATION:

Let's see how we'd answer a single query (K, R).

Since we want to reduce the rank of A_K, it's clearly optimal to only remove elements that are $> A_K$.
Smaller values don't affect its rank at all.

So, suppose there are x elements in $A[1:K-1]$ and y elements in $A[K+1:N]$ that are $> A_K$.
That means the initial rank of A_K is $x+y+1$, and each element we delete will reduce it by 1.

Our target is to make the rank of A_K be $\leq R$.
If $x+y+1 \leq R$ the answer is 0, otherwise we need to delete at least $(x+y+1)-R$ elements.

In particular, we'd like to find the smallest integer z such that deleting at most z elements from both sides allows us to delete at least $x+y+1-R$ elements in total.
Finding this z is not too hard.

- One way is to note that as z increases, we can delete more elements — meaning binary search can be applied.
  If z is fixed, the maximum number of deleted elements is $\min(x,z)+\min(y,z)$, so you can binary search on z to find the first time this number is at least $x+y+1-R$.
  This takes $\mathrm{O}(\log N)$ time.
- Alternately, you can do a bit of casework and find z in $\mathrm{O}(1)$ time.
  it can be seen that:
  - If $y-x \leq R-1$ (assuming $x \leq y$), we'll have $z = \lceil \frac{x+y+1-R}{2} \rceil$
  - Otherwise, we'll have $z = y-(R-1)$.
    These can be derived by considering the cases when you delete $\leq \min(x,y)$ elements from both sides (in which case an equal number can be deleted from both); and when this isn't enough.

At any rate, we can see that the 'slow' part of the query is finding x and y: once they're known, the answer can be found in $\mathrm{O}(\log N)$ or $\mathrm{O}(1)$ time.

Essentially, we want to know the following information:

- For each $1 \leq i \leq N$, how many elements before i are greater than A_i? How many elements after i are greater than A_i?

Let's call these values $\mathrm{left}[i]$ and $\mathrm{right}[i]$, respectively.
All the values of $\mathrm{left}[i]$ and $\mathrm{right}[i]$ can be found in $\mathrm{O}(N \log N)$ time by using an appropriate data structure.

In C++, the simplest way to do this is to use the builtin `__gnu_pbds::tree`, also commonly called "ordered set" or "indexed set".
A blog on the basic functionality and how to include it can be found **here** .
Essentially, this allows us to keep a data structure that supports everything `std::set` does, along with:

- Given k, find the k-th largest element present in the set (`find_by_order`)
- Given an element x, find the number of elements in the set that are $< x$ (`order_of_key`)

both in $\mathrm{O}(\log N)$ time.

For our use-case, the second query type, `order_of_key`, is what we want.

For example, we can calculate the left values by iterating across the array from left to right, each time querying for the number of elements that are $\leq A_i$ (which in turn also tells us how many elements are $> A_i$, namely $\mathrm{left}[i]$).
Then, insert A_i into the set and continue on.

In languages other than C++, various other data structures can fulfill this purpose.
For example, you can use a segment tree/fenwick tree built on values, or write (or copy) a custom self-balancing BST (for example, a treap) augmented with appropriate data (namely, subtree sizes).

TIME COMPLEXITY

$\mathrm{O}(N \log N + Q)$ per testcase.

PROBLEM:

Is it possible to tile a rectangular floor with length L and width W, using only rectangles whose perimeter *isn't* divisible by 4?

EXPLANATION:

Recall that the perimeter of a rectangle whose side lengths are a and b, is $2 \cdot (a+b)$.
For this to be *not* divisible by 4, we'd need $(a+b)$ to be odd — which means exactly one of a and b must be even, and the other must be odd.
In particular, this means the area of such a rectangle (which equals $a \cdot b$) must be even.

Now, let $A = L \cdot W$ be the total area of the floor.
Suppose we tile it with k rectangles, and their areas are $A_1, A_2, \ldots, A_k$.
Of course, since it's a tiling, $A = A_1 + A_2 + \ldots + A_k$ must hold.

For the tiling to be valid, we need each A_i to be even, as discussed earlier.
Notice that this forces A to also be even, as the sum of some even numbers.

So, if the area $A = L \cdot W$ is odd, the answer is immediately `No`.
On the other hand, if A is even, the answer is always `Yes`!
This is because a rectangle with even area can always be tiled using 1×2 (or 2×1) tiles, which have perimeter 6 and so are usable.

TIME COMPLEXITY

$\mathrm{O}(1)$ per test case.

PROBLEM:

You're given an array A. Find the number of non-empty subsets of indices whose removal results in every subarray having even sum.

EXPLANATION:

Suppose we removed some indices, and the array satisfies the given condition, i.e, $\sum_{k=i}^{j} A_k$ is even, for each $1 \leq i \leq j \leq |A|$.

Then, in particular this holds for $i = j$, which means every element must itself be even.
Of course, if every element is even then every subarray sum is also even; so we've reduced our problem to counting the number of ways of removing indices such that all remaining elements are even.

That's not too hard:

- First, *all* odd elements must be removed.
- Suppose this leaves K even elements. We can then choose any subset of them to remove, for 2^K possible ways.

There's one edge case: since the statement asks for **non-empty** subsets, if $K = N$ (i.e all elements of the original array are even), the answer is $2^N - 1$; since we can't choose the empty subset.

Since $K \leq N$, computing 2^K can be done by just repeated multiplication using a loop.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

Starting with the set $S = \{1, 2, \ldots, N\}$, you'd like to reach its subset Q.
To do so, you can perform the following move:

- Choose an integer $1 \le x \le N$, and delete the smallest multiple of x from S.
  The cost of doing so is x.

Find the maximum cost required to turn S into Q.

EXPLANATION:

Let D be the set of elements we need to delete - that is, $D = \{1, 2, 3, \ldots, N\} \setminus Q$.

For each $x \in D$, we need to choose a factor of it to delete it.
Given that our aim is to maximize the cost, obviously the best we can do is to choose x itself!

This means the answer is simply the sum of the elements of D.

Note that D has size $N - M$, and N can be quite large.
So, it's not really possible to iterate across all the elements.

Instead, notice that D and Q together have all the elements $\{1, 2, \ldots, N\}$.
So, we know that

$$sum(D) + sum(Q) = 1 + 2 + \ldots + N$$
$$sum(D) = 1 + 2 + \ldots + N - sum(Q)$$

The elements of Q are given in the input, so computing their sum is easy.
Further, there's the well-known formula $1 + 2 + \ldots + N = \frac{N \cdot (N+1)}{2}$.

So, the answer is:

$$\frac{N \cdot (N+1)}{2} - \sum_{i=1}^{M} Q_i$$

Note that the answer can exceed the range of 32-bit integers, so make sure to use larger datatypes such as `long long` (C++) or `Long` (Java).

TIME COMPLEXITY

$\mathrm{O}(M)$ per testcase.

PROBLEM:

There's a pile with N stones. You also have an integer K.

Alice and Bob play a game, Alice starts first.
On their turn, a player can choose an integer x such that $\gcd(x, K) = 1$ and there are at least x stones remaining; and remove x stones from the pile.
The winner is the person who makes the last move.

Choose whether you play as Alice or Bob; and win the game.

EXPLANATION:

First, we need to decide who we're playing as.
For this, we need to know who wins the game, given N and K.

If $K = 1$ then every move is valid; so Alice wins on the first move by just removing all N stones.
We only need to deal with the $K > 1$ case.

$\gcd$ is a function that depends on prime factorization; so intuitively this should have something to do with the prime factorization of K.

Indeed, if you work out a few small cases on paper, and especially observe which moves are possible for a given K, you might notice the following:

$$\text{Let } p \text{ be the smallest prime factor of } K.$$
$$\text{Alice wins if and only if } N \bmod p \neq 0.$$

That is, Alice wins if and only if p doesn't divide N.
The proof of this also gives us a winning strategy.

The important observation that makes this work is that, if p is the smallest prime that divides K, then $\gcd(K, x) = 1$ for all $1 \leq x < p$.
That is, the moves $1, 2, 3, \ldots, p - 1$ are always valid moves.
This is because $\gcd(K, x) > 1$ means K and x share a prime factor; but if $x < p$ then K would have a prime factor smaller than p, which is a contradiction.

Further, if $N \bmod p = 0$, there's no valid move x such that $(N - x) \bmod p = 0$.
Any such move would require x to be a multiple of p; but then $\gcd(x, K) > 1$ so it's not a valid move.

Together, the above points tell us that:

- If $N \bmod p = 0$, then the current player can only move to a non-zero state.
- If $N \bmod p > 0$, the current player can always move to a zero state; because $N \bmod p < p$, it is enough (and always a valid move) to simply remove $N \bmod p$ stones.

In particular, if $N \bmod p > 0$, the current player will always have a valid move; so they can never be lost.

Putting it all together, we have a fairly simple strategy:

- If $N \bmod p = 0$, play as Bob.
  Otherwise, play as Alice.
- This guarantees that on your turn, $N \bmod p > 0$.
  Simply remove $N \bmod p$ stones, which is always possible.

Finding the smallest prime factor of K can be done in $\mathrm{O}(\sqrt{K})$ time.

TIME COMPLEXITY

$\mathrm{O}(N + \sqrt{K})$ per testcase.

PROBLEM:

You're given a permutation P of $\{1, 2, \ldots, N\}$.
Chef can choose a subarray $[L, R]$ of length strictly less than N and delete it. The *score* of this operation is the length of the subarray.

Find the maximum score of an operation such that the remaining elements still form a permutation.

EXPLANATION:

Suppose Chef deletes subarray $[L, R]$ and the remaining elements form a permutation of $\{1, 2, \ldots, K\}$.
Then, note that the length of the subarray $[L, R]$ must be exactly $N - K$, since it must contain the elements $K + 1, K + 2, \ldots, N$ (in some order) — that's the only way for the remaining parts to form a permutation.

We want to *maximize* $N - K$, which means we want to *minimize* K.
That is, we want to find the smallest integer K such that elements $K + 1, K + 2, \ldots, N$ appear as a contiguous subarray in the permutation.

This gives us a rather straightforward $\mathrm{O}(N^2)$ solution (which is too slow to get AC, but serves as a good starting point):

- Iterate K from 1 to N
- For a fixed K, mark all the positions that contain integers $\geq K + 1$, then check if they form a contiguous subarray.

Now, we'd like to speed up the second part of this algorithm.

Let's define an array pos, such that $\mathrm{pos}_x = i$ if and only if $P_i = x$.
That is, pos denotes the positions of the elements in P.

Then, for a fixed K, we're interested in $\mathrm{pos}_{K+1}, \mathrm{pos}_{K+2}, \ldots, \mathrm{pos}_N$.
In particular,

- Let $L_K = \min(\mathrm{pos}_{K+1}, \mathrm{pos}_{K+2}, \ldots, \mathrm{pos}_N)$ and $R_K = \max(\mathrm{pos}_{K+1}, \mathrm{pos}_{K+2}, \ldots, \mathrm{pos}_N)$.
- Then, $[L_K, R_K]$ is the *unique smallest* subarray that contains all the elements $\geq K + 1$.
- All we need to do is check whether the length of this subarray is exactly $N - K$, i.e, if $R_K - L_K + 1 = N - K$.

Note that L_K and R_K are respectively suffix minimums/maximums of the pos array.
The pos array itself can be computed in $\mathrm{O}(N)$, after which computing its suffix minima/maxima is trivial.
After this, each K is processed in $\mathrm{O}(1)$ time, giving us a solution in $\mathrm{O}(N)$ overall.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case

PREREQUISITES:

Topological sorting

PROBLEM:

Given an array A containing integers from -1 to $N-1$, find the lexicographically permutation P (or claim that none exist) such that for each $1 \le i \le N$ the following condition is true

- If $A_i = -1$, no constraint on P_i
- If $A_i = 0$, then $P_i > P_j$ for every $1 \le j < i$
- Otherwise, if $A_i = k$, then $k < i$ and k is the largest index such that $A_k > A_i$.

EXPLANATION:

Clearly $A_i < i$ must hold for every index, so if this is not the case for some i the answer is immediately -1.

A simpler version

First, let's try to solve the problem for the case when A doesn't contain any -1, i.e, there is a constraint on every i. Also, let's ignore the lexicographic order condition, and just try to find *any* solution.

Note that $A_i = k$ gives us the following pieces of information:

- $P_i < P_k$ (if $k > 0$)
- $P_i > P_j$ for **every** $k < j < i$

Using this information, let's construct a directed graph on N vertices. For each constraint we have of the form $P_i < P_j$, add the directed edge $i \to j$ to this graph.

Looking at this graph tells us a few things:

- If there is a path from i to j in the graph, then $P_i < P_j$ must hold.
- Extending the above point, if the graph contains any cycles then no valid P can exist, since it would mean $P_i < P_i$ for some i
- That means we're left with a directed acyclic graph, which means it has a topological order, say $[u_1, u_2, \ldots, u_N]$.
- Note that simply assigning $P_{u_i} = i$ gives us a valid permutation!

We now have a solution to the simpler version of the task. There are three issues with it:

- First off, the graph we create can have $\mathrm{O}(N^2)$ edges since each A_i can create $\mathrm{O}(N)$ edges. This is too slow.
- Second, we found *some* solution, not the lexicographically smallest one
- Third, we need to deal with the existence of -1 in A.

Let's deal with all three of those separately.

The full solution

Dealing with -1

This is by far the simplest issue to resolve: note that $A_i = -1$ means this index doesn't give us a few constraints, so we simply ignore it and move on.

Lexicographic order

Once we create our directed graph, notice that *every* possible solution corresponds to some topological order of this graph.

In other words, what we want to do is find the lexicographically smallest topological order of the given DAG. This can be done with a simple extension of one of the standard topological sort algorithms.

One of the ways to compute a topological sort is to compute the indegrees of every node, then keep picking a node with zero indegree and update its neighbors.
When doing this process, simply pick the *smallest* available vertex at each step: this gives us the lexicographically smallest topological order.
Reducing edges

Creating $\mathcal{O}(N^2) edges is of course going to lead to TLE, so we'd like to reduce their number while ensuring that no information is lost.

Note that each edge we create is one of two types:

- $i \to A_i$, 'leftward' edges
- $j \to i$ where $A_i < j < i$, 'rightward' edges

The leftward edges are $\mathrm{O}(N)$ in number so we can leave them as is; it's the rightward ones that there can be many of.
Do we really need all of them? Let's analyze them a bit.

Suppose we had edges $i \to j$ and $i \to k$, where $i < j < k$. What does this mean?

- We have the constraints $P_i < P_j$ and $P_j < P_k$
- However, the only possible way the $i \to k$ edge can exist is if $A_k < i$
- In particular, this means that $P_j < P_k$ must also hold, so we'd have the edge $j \to k$
- Notice that this means that we don't actually need the $i \to k$ edge: the path $i \to j \to k$ already exists to capture that information.

In other words, we only need at most one rightward edge per index! This reduces the number of edges to $\mathrm{O}(N)$.

Putting everything together for a complete solution, we have the following approach:

- Iterate i from 1 to N. Let S denote the set of indices that don't have a rightward edge yet.
- If $A_i = -1$, ignore it
- Otherwise, add the following edges:
  - $i \to A_i$
  - $j \to i$ for every j such that $j \in S$ and $A_i < j < i$
  - For every $j \to i$ edge added, remove j from S
- Insert i into S
- Finally, compute the lexicographically smallest topological sort of the resulting graph to obtain the answer.

TIME COMPLEXITY:

$\mathrm{O}(N \log N)$ per testcase.

PROBLEM:

N people are lost in a forest, each with their own radios.
Person i's radio can transmit K_i frequencies, given as a list $F_i = [f_{i,1}, f_{i,2}, \ldots, f_{i,K_i}]$

Two people a and b can communicate with each other if they either share a common frequency (i,e $F_a \cap F_b \neq \emptyset$), or there exists a person c who can communicate with both of them.

In one operation, you can move a frequency from one person's radio to another.
That is, choose i and j, and an element $x \in F_i$. Then, delete x from F_i and append x to F_j.

Find the minimum number of operations to reach a state where everyone can communicate with everyone else; or claim that it is impossible.

EXPLANATION:

The condition on two people being able to communicate sounds suspiciously similar to connectivity in a graph (two vertices are connected if they either have a direct edge between them, or if there's some other vertex that is connected to both of them).
This should hint you towards thinking about graphs.

Let C be the total number of distinct frequencies present in the input.
Consider an undirected *bipartite* graph on $N + C$ vertices, where:

- There are N vertices on the left, representing the people
- There are C vertices on the right, representing the frequencies
- There's an edge between i on the left and j on the right if person i's radio can transmit frequency j.

This graph has exactly $\sum_{i=1}^{N} K_i$ edges, so creating and storing it isn't an issue since the sum of K_i across all testcases is small enough.

Now, note that person i and person j can communicate with each other if and only if they lie in the same connected component of this graph.

Proof

Suppose person a and person b are in the same component. Then, there must exist a path $a = x_1 \to y_1 \to x_2 \to y_2 \to \ldots \to x_{k-1} \to y_{k-1} \to x_k = b$ of *even length* (since the graph is bipartite), where each x_i is a person and each y_i is a frequency.

Via this path, clearly for each $1 \le i < k$ person x_i can communicate with person x_{i+1} via frequency y_i, since they can both transmit it.
Chaining this relation allows person a to communicate with person b.

Conversely, suppose person a and person b can communicate with each other.
Then, either a and b can communicate directly (in which they're surely in the same component), or there must exist a sequence of people $x_1, x_2, \ldots, x_k$ such that:

- a can communicate with x_1 directly (i.e they share a frequency)
- x_i can communicate with x_{i+1} directly for each i
- x_k can communicate with b directly

Putting all this together means a and b must lie in the same component, as claimed.

So, our aim is to ensure that all N people end up in the same component.

As for the operation we perform, moving a frequency from one person to another translates to shifting the 'left' endpoint of the corresponding edge, while the 'right' endpoint remains the same.

Let's look at how moving an edge like this affects connectivity.
Suppose we move frequency x from person i to person j, i.e, replace the edge $i \leftrightarrow x$ with $j \leftrightarrow x$.
This can be thought of as deleting one edge and adding a new one.
Note that:

- Removing an edge either keeps the number of components the same, or increases it by 1
  In particular, if the edge is a bridge (doesn't lie on a cycle), the number of components will increase by 1; otherwise the number of components will not change.
- Adding an edge either keeps the number of components the same, or decreases it by 1
  In particular, if the edge joins vertices from two different components then the component count reduces by 1; else it doesn't change.

So, an operation is only 'useful' (i.e, decreases the number of components) if the removal stage keeps the number of components the same, while the addition stage decreases it by 1.
That is, an operation must take an edge that lies on a cycle and use it to join two different components.

The second part is quite simple: we're free to choose the left endpoint, so we can always choose a vertex not in the current component.
Dealing with the first part is seemingly trickier, but is in fact not that hard either.

Consider some connected component with X vertices and Y edges.
Fix a spanning tree of this component, which uses $X - 1$ of these Y edges.
This leaves us with $Y - (X - 1)$ 'free' edges, which can certainly be used to connect to other components.

Suppose there are W components in total, and let f be the total number of free edges across all these components.
If $f \geq W - 1$, we can directly use $W - 1$ free edges to connect all the components (and doing it in less than $W - 1$ operations is clearly impossible).
On the other hand, if there are less than $W - 1$ free edges then it's impossible to connect all the components!

Proof

Suppose $f < W - 1$.
Then, the total number of edges we have is $N + C - W + f$. This is because:

- There are $N + C$ vertices.
- A spanning forest of these vertices with W components uses exactly $N + C - W$ edges.
- Then, there are f free edges.

If $f < W - 1$, then $N + C - W + f < N + C - W + W - 1 = N + C - 1$, i.e, there are strictly less than $N + C - 1$ edges in total.
It's impossible to connect $N + C$ vertices with less than $N + C - 1$ edges.

Note that if all N people are in the same component, then the graph as a whole must be connected because we can't change the right endpoints of the edges (so each frequency is connected to at least one person, and hence to all the people).
This means we'd like to end up with a connected graph on $N + C$ vertices, but simply can't.

This gives us a pretty simple solution:

- Construct the graph mentioned above, and compute W and f: its number of components and free edges.
- If $f \geq W - 1$, the answer is $W - 1$.
  Note that this is equivalent to checking whether the number of edges is at least $N + C - 1$, so in fact there's no need to even compute the free edges.
- Otherwise, it's impossible.

Counting components can be done in linear time with DFS/BFS (or use a DSU for *technically* worse complexity but same speed in-practice), so we're done.

TIME COMPLEXITY

$\mathrm{O}(N + \sum K_i)$ per test case.

PROBLEM:

Given an integer K in base P, find the smallest integer $X \geq 0$ such that $K + X$ contains all the digits $0, 1, 2, \ldots, P - 1$ in base P.

EXPLANATION:

This is more of an implementation task than anything else, but being a bit careful can lead to a relatively short and clean implementation.

Instead of finding the smallest value of X, let's find the smallest possible value of $K + X$; in the end, we can obtain X by subtracting K from it.

First, note that $K + X$ will contain either P or $P + 1$ digits: at least P since we need P distinct digits, and at most $P + 1$ because $N \leq P$ means we never need to go any higher.
In particular, for the answer to have P digits, all of its digits need to be distinct.

Now let's get rid of a couple of simple edge cases:

- If $N < P$, then the optimal value of $K + X$ is $[1, 0, 2, 3, 4, \ldots, P - 1]$; this is the smallest integer with all the digits.
- Now we're left with $N = P$ always.
- The largest possible P-digit number with distinct digits is $[P - 1, P - 2, \ldots, 2, 1, 0]$. If the given number is larger than this (which can be checked by comparing them lexicographically), then we *must* use $P + 1$ digits, and the answer is $[1, 0, 0, 2, 3, 4, \ldots, P - 1]$; this is the smallest $P + 1$-digit number containing all digits.

Once those cases are out of the way, we know that it's always possible to achieve a P-digit answer; so our task is to make all the digits distinct.

To minimize $K + X$, we'd like to keep as large a prefix of K unchanged as possible.
So, let's iterate i from 1 to P.

- if A_i hasn't appeared before, we don't need to increase this index right now
- If A_i has appeared before, we *must* change some index $\leq i$ in order to not have repeated digits.

Let x be the first time we meet a repeated digit (if there are no repeated digits, the answer is obviously 0 so we only focus on the case when a valid x exists).

We have to increase some digit at a position $\leq x$. So, let's do the following:

- Let's check if we can replace A_x with something higher than it. This is only possible if there exists a digit $> A_x$ that has *not* appeared before position x (recall that our aim is distinct digits).
- If we can't replace A_x with something larger than it, instead check for position $x - 1$, then position $x - 2$, and so on. One of these will definitely satisfy the condition since we know a P-digit answer is possible.
  - One simple way of quickly checking whether replacement is possible is to maintain a set of unseen values and find its maximum. When moving from x to $x - 1$, insert A_{x-1} into this set of unseen elements.
  - This allows each position to be processed in $\mathrm{O}(\log N)$ time, for $\mathrm{O}(N \log N)$ overall. You can also maintain a boolean `mark` array and a pointer to it for $\mathrm{O}(N)$ time.
- Suppose we've found the position y that needs to be increased. Do the following:
  - Set A_y to the smallest unseen element larger than it
  - Then, fill positions $y + 1$ to P with the remaining unseen elements in ascending order.

Now we have the array representing $K + X$, so we need to compute $X = (K + X) - K$.
Notice that we only need X modulo $10^9 + 7$.

So, compute both K and $K + X$ modulo $10^9 + 7$ (for example, using binary exponentiation to quickly compute powers modulo the mod), and subtract one from the other to obtain the final answer.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Basic combinatorics

PROBLEM:

There are N asteroids in the 2-D plane, the i-th initially at (X_i, Y_i).
Each second, every asteroid moves one step left.

An optimal strategy to destroy the asteroids is to pick an asteroid with the smallest X-coordinate each second.
A random strategy is to pick a random asteroid each second.

Find the probability that a random strategy is optimal.

EXPLANATION:

Let's number the asteroids $1, 2, \ldots, N$.
Then, any strategy of destroying the asteroids can be described by an arrangement of the numbers 1 to N, i.e, a *permutation* P of length N which tells us that the i-th asteroid destroyed was P_i.

The total number of strategies is thus $N!$, and a random strategy simply picks one out of these $N!$ uniformly at random.
So, if there are K *optimal* strategies in total, the probability that a random strategy is optimal is exactly $\frac{K}{N!}$
This means all we need to do is compute K, the number of optimal strategies.

We can make the following observations:

- The y-coordinates of the asteroids don't matter at all, they neither change nor affect which one is chosen. They can be entirely ignored.
  This makes our problem essentially one-dimensional, we only care about the x-coordinates.
- All the asteroids move similarly, so their relative positions stay the same. That is, if asteroid i is initially to the left of asteroid j, then this relation will continue to hold no matter how much time passes.
  In other words, we can ignore the passage of time entirely, which makes finding an optimal strategy obvious.

Let $x_1, x_2, \ldots, x_k$ be the *distinct* x-coordinates that appear among the points, with $x_1 < x_2 < \ldots < x_k$.
Let c_i be the number of points whose x-coordinate is x_i.

Then, an optimal strategy must look as follows:

- First, pick all c_1 points whose x-coordinate is x_1. These c_1 points can be chosen in any order, for $c_1!$ ways in total.
- Next, pick all c_2 points whose x-coordinate is x_2. These c_2 points can be chosen in any order, for $c_2!$ ways in total.

  $\vdots$
- Finally, pick all c_k points whose x-coordinate is x_k. These c_k points can be chosen in any order, for $c_k!$ ways in total.

The number of optimal strategies is thus simply the product of all the above values, i.e, $\prod_{i=1}^{k} c_i!$

If factorials are precomputed, computing the above product becomes simple.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Greedy algorithms, binary search

PROBLEM:

You're given a string S. In one move, you can pick and index i and replace S_i with either $S_i + 1$ (upwards move) or $S_i - 1$ (downwards move), where the values are cyclic from a to z.

You can perform at most P upwards moves and at most Q downwards moves.
What's the lexicographically smallest string you can obtain?

EXPLANATION:

Since our objective is lexicographic minimization, the obvious choice is to try and make the first character a, then make the second character a, and so on.

Let's try to find the largest prefix of S that can be turned into a's.

Suppose we want to check whether the first k characters of S can be turned into a.
First, compute the values up_i and $down_i$ for each $1 \le i \le k$, where up_i is the number of upward moves required to turn S_i into a, and $down_i$ is defined similarly.

Now, we want to choose up_i for some indices and $down_i$ for the rest, such that the sum of chosen up_i is at most P, and the sum of chosen $down_i$ for the rest is at most Q.

This can be done greedily!
That is, sort the pairs $(up_i, down_i)$ in ascending order of up_i value. Then, greedily keep taking the smallest up_i values as long as you don't exceed P, after which you take the $down_i$ values for everything else.
Finally, check if the sum of taken $down_i$ values is $\le Q$.

Proof of correctness

The key to why this works is the fact that up_i and $down_i$ are complementary.
That is, $up_i + down_i = 26$ (unless $S_i = \mathrm{a}$, in which case $up_i = down_i = 0$ and we can just ignore this index).

This means that smaller values of up_i correspond to larger values of $down_i$, and vice versa.
So, if we sort the pairs and have $up_1 \le up_2 \le \ldots \le up_k$, that also means we'll have $down_1 \ge down_2 \ge \ldots \ge down_k$.

Intuitively this already makes it obvious why taking a prefix of the up_i values is good, since it'll make us take a suffix of the $down_i$ values which is the best we can do there too.

It's also possible to prove this algebraically, using an exchange argument.
Suppose we choose up_x, but don't choose up_y where $y < x$ (in sorted order); meaning we chose $down_y$.

Suppose we replaced up_x with $down_x$ and $down_y$ with up_y.
Then, since $up_x \ge down_x$ and $down_y \ge up_y$, this reduces (or rather, doesn't increase) *both* the sum of up_i values and the sum of $down_i$ values, which is obviously good.
Repeating this replacement operation makes us end up with a prefix of up values and a suffix of $down$ values, as claimed.

So, we can now check whether the first k characters can be all turned into a, in $\mathrm{O}(k \log k)$ time. It's possible to implement this in $\mathrm{O}(k + 26)$ time, but unnecessary to get AC.
Our aim is to find the maximum such k.

Notice that if the first k characters can be turned into a, so can the first $k-1$ characters.
So, we can use binary search!
This allows us to find the largest valid k in $\mathrm{O}(N \log^2 N)$ time.

In addition, note that our method also tells us how many moves are remaining of each type, since it minimizes *both* the number of upward and downward moves.
We need to use these remaining moves to deal with the positions from $k+1$ onwards.

First, we know that S_{k+1} *cannot* be made into a, so the best we can do is to perform downward moves on it as much as possible. This uses up all our downward moves.

Finally, for each index i from $k+2$ to N, check whether S_i can be turned into a using the remaining upward moves; and if it can, do so.

It is also possible to implement this without binary search, by maintaining a set/priority queue of lowest up_i values and removing the largest of them whenever the sum exceeds P.

In fact, it's even possible to implement this solution in $\mathrm{O}(N)$; doing so is left as a challenge to the reader 😄

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case

PROBLEM:

The *goodness* of a sequence S is defined as follows:

- Sort S in increasing order;
- Then, its goodness is the number of indices i such that $S_i = i$.

Given an array A, find the sum of goodness across all its subsequences.

EXPLANATION:

First, let's try to solve this problem for when all the elements of A are **distinct**.

For this, we can use the technique of looking at *contribution*.
That is, instead of finding the sum of goodness across all sequences, we'll find for each element the number of sequences it contributes to the goodness of.

Since all the elements are distinct, this is not too hard.
Let's consider an element x in A, we want to count the number of subsequences of A such that $S_x = x$.
For this:

- There should be exactly $x - 1$ elements less than x, to be placed at indices $1, 2, \ldots, x - 1$.
- There can be any number of elements $> x$, since they don't affect the position of x in the sorted subsequence.

This then turns into a rather simply counting problem:

- If there are L elements less than x in A, we can choose $x - 1$ of them in $\binom{L}{x-1}$ ways.
- If there are R elements greater than x in A, we can choose any subset of them freely in 2^R ways.
- Multiplying them together, we get $\binom{L}{x-1} \cdot 2^R$ ways.

Finding L and R isn't very hard.
Let's sort the array A, then if $A_i = x$ in this sorted array we have $L = i - 1$ and $R = N - i$.

So, after sorting, we can solve the problem in $\mathrm{O}(N)$: for each index i, add $\binom{i-1}{A_i-1} \cdot 2^{N-i}$ to the answer.

Computing binomial coefficients quickly can be done by precomputing factorials, for example as described **here**.

Now, what happens if the elements aren't necessarily distinct?
In fact, the exact same solution works!
That is, the answer is simply

$$\sum_{i=1}^{N} \binom{i-1}{A_i - 1} \cdot 2^{N-i}$$

after A is sorted.

Proof

When elements aren't distinct, the only real issue that we face is that which element is placed at which position in sorted order isn't uniquely determined; which might throw off our contribution counting.
For example, the subsequence $[1, 1]$ should be counted against exactly one of the ones; not both of them.

One way to resolve this ambiguity is to uniquely define an order among equal elements as well.
For example, if we have k ones in A, we can label them $[1_1, 1_2, 1_3, \ldots, 1_k]$, and we can say that if 1_i and 1_j are both chosen in a subsequence (where $i < j$), then 1_i will be placed before 1_j in sorted order.

This immediately resolves the ambiguity, since if $S_i = i$ for some subsequence, it will contribute only towards one *specific* i (depending on the labels of the i's chosen for this subsequence).

Now, note that because we have a uniquely defined order among all the elements, they're all essentially distinct! So, our initial solution, where we treated elements as distinct, works here too.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

Dynamic programming, range-max queries

PROBLEM:

You're given an array S. For Q queries of (L, R), answer the following:

- For a fixed value of D, you start from L with a score of D and move towards R. At each index, $S_i \leq D$ must hold. Further, if $S_i = D$, reduce D by 1.
  Find the smallest value of D that allows you to reach R from L.

EXPLANATION:

Let $M = \max(S_L, S_{L+1}, \ldots, S_R)$.
Clearly, the answer to query (L, R) must be at least M, since we will encounter an M along our way and need to be $\geq M$ when we do.

Further, it's also easy to see that $D = M + 1$ always allows us to reach R, since it's strictly larger than anything in this range and hence will never reduce.
So, our task reduces to checking whether $D = M$ allows us to reach R or not.

Let's see how this process would go.

- Let $i_1 \geq L$ be the first index such that $S_{i_1} = M$. We will definitely reach this index, after which we'd be left with $D = M - 1$.
- From here, we move right till we
  - Reach R, at which point we're done
  - Reach an occurrence of something $> D$, at which point we can't continue
  - Reach another occurrence of D, at which point we move to $D - 1$; and then this process repeats.

This is easy to simulate in $\mathrm{O}(N)$, but we need to do better.

Notice that once we reach the first position where $S_i = M$, the process is rather uniquely defined, because we can pretend we start there with $D = M$.

So, let's find for each index i, the furthest we can travel if we start from i with value S_i.
Let this be dp_i.

dp_i can be computed as follows:

- Let $j > i$ be the first index such that $S_j \geq S_i$
- Let $k > i$ be the first index such that $S_k = S_i - 1$.
- Then, $dp_i = \min(j - 1, dp_k)$, because we can't cross index j no matter what; and if we reach k we're essentially starting from there instead, so we can't go beyond dp_k either.

dp_i can be precomputed quite easily for all indices by iterating i from N down to 1. The only things that need to be taken care of are quickly computing indices j and k.

- Computing k is easy. Since we're iterating in reverse anyway, simply keep a record (using, say, a map) of the last occurrence of every element. Let this be $\text{last}[x]$; then $k = \text{last}[S_i - 1]$.
  Note that after processing index i, you should set $\text{last}[S_i] = i$.
- Computing j needs a bit more effort, but in the case of this specific problem, can be actually done easier.
  Note that rather than finding the next element that's $\geq A_i$, it suffices to find the next element that's equal to

S_i, i.e, $\text{last}[S_i]$ itself. Do you see why?

At any rate, by maintaining the last map, all the dp values can be computed quickly.

After this, for a query (L, R) we need to:

- Find the maximum M on the range $[L, R]$, which is a standard data structure task (use a segment tree/sparse table/whatever).
- Find the leftmost occurrence of this maximum, which is also not too hard: for example, the segment tree/sparse table merge function can be modified slightly to return this, by maintaining pairs of (value, index).
- Let i be the index we found. Then, if $dp_i \geq R$ the answer is M, otherwise the answer is $M + 1$.

TIME COMPLEXITY

$\text{O}((N + Q) \log N)$ per test case.

PREREQUISITES:

Sorting

PROBLEM:

You are given N, M, and an array B of size $N \times M$ of distinct integers

Consider an $N \times M$ grid A whose values form a permutation of B. The following process is carried out:

- A token is placed at $(1, 1)$ and moved towards (N, M) either right or down each step.
- On odd moves (1-st, 3-rd, etc) the token is moved towards its larger neighbor
- On even moves the token is moved towards its smaller neighbor
- The value of the grid taken equals the difference between the maximum and minimum element on the path.

Construct A such that this value is minimized.

EXPLANATION:

First, let's sort B so that $B_1 < B_2 < \ldots < B_{NM}$.

Without loss of generality, assume $N \leq M$.

Note that any path from $(1, 1)$ to (N, M) where we move only right and down will visit exactly $N + M - 1$ cells. Let $L = N + M - 1$ be this length.

Now, ideally we'd like the values on our path to be L consecutive values in B, since this is how we minimize the difference between the largest and smallest of them. However, we also need to account for actually being to place these values to form a valid path, so we can't just choose any subarray of length L.

The less constraints we have, the better.
So, let's try to make our path go down first (recall that $N \leq M$), and then go right after it hits the last row. Once we hit the last row, there's only one way to go so the maximum/minimum conditions don't matter anymore and we can freely place anything there, so we only need to worry about the going down part.

Specifically, we need to ensure we choose values such that:

- $A_{2,1} > A_{1,2}$
- $A_{3,1} < A_{2,2}$
- $A_{4,1} > A_{3,2}$

  $\vdots$

Only by doing so can we ensure we move down for the first N steps.
Notice that the value of $A_{1,1}$ also doesn't matter whatsoever.

In particular, we have $x = \lfloor \frac{N}{2} \rfloor$ constraints of the form $A_{i,1} > A_{i-1,2}$ and $y = \lfloor \frac{N-1}{2} \rfloor$ of the form $A_{i,1} < A_{i-1,2}$.

Of course, the best way to fulfill these is to pair the largest x elements among our chosen ones with the smallest x *unchosen* elements; and the smallest y chosen elements with the largest y unchosen ones.

In order to fulfill these conditions, we also need to ensure that the smallest x unchosen elements are all smaller than our largest x elements; and the largest y unchosen elements are larger than our smallest y.
In particular, this means that if we pick the subarray $[i, i + L - 1]$ then it must satisfy $i > x$ and $i + L - 1 + y \leq N \times M$.

The minimum difference is thus the minimum value of $B_{i+L-1} - B_i$ across all subarrays that satisfy the above conditions on i, which is easy to find by just iterating across B.

Once you've found i, the starting point of the subarray, constructing A is not very hard:

- Place the x largest elements of the chosen subarray at positions $A_{2,1}, A_{4,1}, A_{6,1}, \ldots$
- Place the elements $B_1, B_2, \ldots, B_x$ at positions $A_{1,2}, A_{3,2}, \ldots$
- Place the y smallest elements of the chosen subarray at $A_{3,1}, A_{5,1}, \ldots$
- Place the elements $B_{NM}, B_{NM-1}, \ldots$ at positions $A_{2,2}, A_{4,2}, \ldots$
- Place all the remaining elements of the subarray at $A_{1,1}$ and $A_{N,2}, A_{N,3}, \ldots, A_{N,M}$ in any order
- Place all the unplaced elements in the grid in some order

After sorting B, the rest of the solution is $\mathrm{O}(NM)$, giving us a solution in $\mathrm{O}(NM \log NM)$.

TIME COMPLEXITY:

$\mathrm{O}(NM \log NM)$ per testcase.

PREREQUISITES:

Sorting, binary search

PROBLEM:

There are N problems; Chef needs A_i minutes to solve the i-th one and will afterwards take a break of B_i minutes.
If he has K minutes of free time, what's the maximum number of problems he can solve?

EXPLANATION:

Let's call a problem *fully solved* if Chef both solves it and completely exhausts its break time, and *partially solved* if Chef solves it but does not exhaust its break time before K minutes are up.

It should be obvious that there can be at most one partially solved problem.
Further, if there is a partially solved problem, it will always be the last problem Chef solves.

Now, if Chef fully solves problem i, it will take $A_i + B_i$ seconds. In particular, the order of fully solved problems doesn't matter at all: only their A_i and B_i values since the total time taken is just their sum.
This means that ideally, we fully solve problems in increasing order of $A_i + B_i$.

So, sort the problems in increasing order of $A_i + B_i$.

Now, we have two cases: either there is a partially solved problem, or there isn't.
Treat them separately and take the best answer.

No partially solved problem

Since there is no partially solved problem, if Chef solves i problems the time taken is simply $A_1 + B_1 + A_2 + B_2 + \ldots + A_i + B_i$.

Find the highest i such that this sum is $\leq K$; this can be done by simply iterating across the sorted array of $A_i + B_i$ values.

There is a partially solved problem

There are a couple of different ways to solve this case. Here's one method using binary search (though there exist ways to do it without binary search as well).

Suppose we fix problem x to be the partially solved problem. Problem x then takes up only A_x time; the B_x minutes of resting time can be ignored entirely.

Now, we need to choose some problems to solve fully, before problem i.
We can choose as many as we want, as long as their $A_i + B_i$ values are at most $K - A_x$.

Once again, this can be done greedily: go from smallest to largest $A_i + B_i$ values, pick as long as the sum is within the limit. Just make sure to ignore $A_x + B_x$, since we're fixing it to be partially solved.

In particular, note that we take a prefix of the sorted $A_i + B_i$ values (except maybe one element, if $A_x + B_x$ belongs to this prefix).
That is, if we consider the prefix of length j, its value is $A_1 + B_1 + A_2 + B_2 + \ldots + A_j + B_j$, maybe minus $A_x + B_x$.

This is a non-decreasing sequence, so we can find the optimal j in $\mathrm{O}(\log N)$ by binary searching!
The only thing that needs to be done is to compute prefix sums of the (sorted) $A_i + B_i$ values, and check whether $A_x + B_x$ lies in the given prefix (which is easy since you know x, just check if $x \leq j$).

This allows us to answer a fixed x in $\mathrm{O}(\log N)$. Try every possibility of x for $\mathrm{O}(N \log N)$ time.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PROBLEM:

You have an array A containing N integers.
You're allowed to swap A_i and A_j if $A_i \ \& \ A_j = 0$.

Find the lexicographically smallest array you can obtain.

EXPLANATION:

Let's start with a slow solution first.

Consider an undirected graph on N vertices, where i and j are joined by an edge iff $A_i \ \& \ A_j = 0$.
Let's look at a connected component of this graph, say containing vertices $\{v_1, v_2, \ldots, v_k\}$. Let $v_i < v_{i+1}$.

Clearly, no matter how we perform swaps, these k vertices can only be permuted within themselves.
So, the best we can hope for is that within this component, elements are sorted.
That is, the smallest A_{v_i} value among them is placed at v_1, the second smallest is placed at v_2, and so on.

In fact, it is possible to achieve this!

Proof

Suppose A_i and A_j are in the same component.

Then, there's a path between them, say $A_i, x_1, x_2, \ldots, x_k, A_j$.

Perform the following sequence of swaps:

- Swap A_i and x_1. The order is now $[x_1, A_i, x_2, \ldots, x_k, A_j]$.
- Swap x_1 and x_2. The order is now $[x_2, A_i, x_1, x_3, \ldots, x_k, A_j]$.
- Swap x_2 and x_3, then x_3 and x_4, and so on till you can swap x_k and A_j.
  The order is now $[A_j, A_i, x_1, x_2, x_3, \ldots, x_k]$.
- Now do this in reverse, swapping x_k with x_{k-1} and so on till you swap x_1 and x_2.
  The order is now $[A_j, A_i, x_2, x_3, \ldots, x_k, x_1]$.
- Finally, swap A_i with x_1.
  Now, A_i and A_j have swapped positions, and everything else is where it started; as we wanted!

Since we can perform arbitrary swaps within a component, any permutation is achievable.

Now, the above solution is obviously too slow since there can be $\mathrm{O}(N^2)$ edges.

However, notice that we don't really care about the graph we had: we only care about its *connected components*.
In particular, if we could find an equivalent graph that had far fewer edges but preserved connectivity relations, we'd be done.

Here's one way to accomplish that.

Let $\neg x$ denote the integer $(2^{20} - 1) \oplus x$, i.e, the 20-bit number that has a bit set iff x doesn't.
In particular, since we're dealing with 20-bit integers, $x \ \& \ y = 0$ iff x is a submask of $\neg y$ (and vice versa).

Consider a *directed* graph on $N + 2^{20}$ vertices, where vertices $0, 1, 2, \ldots, 2^{20} - 1$ represent *values* and vertices $2^{20}, 2^{20} + 1, \ldots, 2^{20} + N - 1$ represent the N *indices*.

Create the following edges:

- For each index i, the edge $i \to \neg A_i$

- For each index i, the edge $A_i \to i$
- For each value x, the edge $x \to y$, where y is a bitmask obtained by removing exactly one bit from x.
  For example, if $x = 7$, the edges created here would be $7 \to 6$, $7 \to 5$, $7 \to 3$.
  This ensures that there's a directed path from x to all of its submasks.

Interestingly, this graph achieves what we want: A_i and A_j lie in the same component of the original graph iff indices i and j lie in the same **strongly connected component** of our new directed graph.

Proof

Suppose $A_i \ \& \ A_j = 0$ (which means A_j is a submask of $\neg A_i$).
Then, the graph we created has the path $i \to \neg A_i \to A_j \to j$.
Similarly, the path $j \to \neg A_j \to A_i \to i$ will also exist; which means that i and j lie in the same SCC.

Now, if A_i and A_j were in the same component of the original graph, that means there's a path connecting them consisting of swappable elements.
Using the above argument, It's easy to extend that to a path from i to j (and vice versa) in our digraph, which means once again i and j lie in the same SCC.

Conversely, suppose a path $i \to j$ exists.
Let P be a shortest path from i to j.
There are two possibilities: either P doesn't contain any other index, or it does.

Suppose P doesn't contain any other index.
Then, note that by construction, the only outgoing edge from i is $i \to \neg A_i$, and the only incoming edge to j is $A_j \to j$.
This means P is a path from $\neg A_i$ to A_j, that doesn't touch any other index.
However, the way we constructed the graph on values, each vertex can only reach its submasks.
So, A_j must be a submask of $\neg A_i$, which means $A_i \ \& \ A_j = 0$.

Next, suppose P does contain other indices.
Let the indices we visit be, in order, $i \to x_1 \to x_2 \to \ldots \to x_k \to j$
The above argument tells us that each adjacent pair of elements here has bitwise AND 0.

So, i and j were in the same component of the original graph as well, and we're done!

Our digraph has $2N + 2^{20} \cdot 10$ edges, which is small enough that a linear-time SCC algorithm should run in time.

If your implementation uses too much memory (especially, many small lists) it can affect runtime as well.
One optimization which allows for a decently large speedup is to just not store most edges.
The edges of the form $x \to submask(x)$ form the bulk of the edges, but they have a rather nice structure, so instead of storing them, they can simply be created and used when processing the appropriate vertex.

TIME COMPLEXITY

$\mathrm{O}(N \log N + 2^{20} \cdot 10)$ per testcase.

PROBLEM:

You are given a permutation P.
In one move, you can swap P_i and P_j for a cost of $|i-j|$.

Find the minimum cost to sort P, and a sequence of swaps that achieves this minimum.

EXPLANATION:

First, let's try to find the minimum cost.

For each $1 \le i \le N$, define d_i to be the *signed distance* that element i needs to travel to reach position i. The sign is negative if it needs to move left and positive otherwise.
For example, if $P = [3, 2, 1]$, then $d_1 = -2$, $d_2 = 0$, $d_3 = 2$.

In particular, if $P_x = i$, then $d_i = i - x$.
Note that for any permutation P, we have $\sum_{i=1}^{N} d_i = 0$.
This tells us that the sum of positive d_i equals the (negative of) the sum of the negative d_i.

So, let S be the sum of positive d_i, i.e, $S = \sum_{i=1}^{N} \max(0, d_i)$.

It turns out that the minimum cost is exactly S.

Proof of lower bound

Suppose we swap P_i and P_j. Let $P_i = x$ and $P_j = y$, and $i < j$.

When we perform this swap, it can be seen that d_x *decreases* by $j - i$, while d_y *increases* by $j - i$.

Note that, in particular, we can only decrease at most one positive d_x in each move.
This means that a lower bound on the cost is clearly the sum of all positive d_x, which is S.

So, if we are able to construct a sequence of swaps that achieves a cost of S, we'll be done.

There are several ways to achieve this, here's one using the d_i from above.
The idea is to try and move elements left while we can; while always reducing positive d_i and increasing negative ones.

While the permutation is not sorted, repeat the following:

- Compute the d_x values for each x.
- Find the smallest *index* i such that $d_i < 0$.
- Find the largest *index* j such that $j < i$ and $d_j > 0$ (such a j will always exist)
- Perform the swap on positions i and j.

Each step takes $\mathrm{O}(N)$ time since we run a couple of loops, and we do this $\mathrm{O}(N^2)$ times in total, so the algorithm runs in $\mathrm{O}(N^3)$ time.

TIME COMPLEXITY:

$\mathrm{O}(N^3)$ per testcase.

PREREQUISITES:

Combinatorics

PROBLEM:

Given a multiset A, count the number of its subsets whose mode is unique.

EXPLANATION:

For each x, let freq_x denote the frequency of x in the multiset.

Most counting problems require us to fix some property of the object we're counting, and this one is no different. Since we'd like to count the number of subsets with unique mode, the defining factors of a subset are:

- Which element is the mode, and
- The frequency of the mode in this subset

So, let's fix both of these: suppose x is the mode of the subset and occurs k times in this subset; and let's attempt to count the number of valid subsets.

Note that, for any *other* integer y:

- If $\text{freq}_y < k$, then there are $(\text{freq}_y + 1)$ choices for y: it can occur 0 times, 1 time, 2 times, $\ldots$ freq_y times; and all these choices are viable since they don't affect the mode
- If $\text{freq}_y \geq k$, then y has k choices: it can appear anywhere between 0 and $k-1$ times.

These are all independent choices for each y, and so the total number of subsets equals the product of:

- $\prod(\text{freq}_y + 1)$ across all y whose frequency doesn't exceed $k-1$; and
- k^{m-1}, where m is the number of elements whose frequency in A is $\geq k$ (we subtract 1 because x also has frequency $\geq k$ but shouldn't be counted here).

Now, notice that:

- $\prod(\text{freq}_y + 1)$ is pretty much just a prefix product, if frequencies are sorted in ascending order.
- The number of frequencies that are $\geq k$ can also be computed quite easily if the sorted list of frequencies is known (either directly via binary search, or in $\text{O}(1)$ by precomputing some suffix sums).

In particular, we can calculate the number of subsets for the pair (x, k) in $\text{O}(1)$ or $\text{O}(\log N)$ after a bit of precomputation.

This is enough to solve the problem: simply iterate over all pairs of (x, k) (such that $k \geq 1$) and perform this computation; the answer is the sum of all these values.

The number of pairs we consider is exactly $\text{freq}_1 + \text{freq}_2 + \text{freq}_3 + \ldots + \text{freq}_{10^9} = N$, so the time complexity of this is $\text{O}(N \log N)$.

TIME COMPLEXITY

$\text{O}(N \log N)$ per testcase.

PREREQUISITES:

Greedy algorithms or **topological sorting**

PROBLEM:

You're given a string S of length N.
You can swap adjacent characters of it as long as the characters differ by at most one (i.e, a and b, b and c, $\ldots$, y and z).

Find the lexicographically smallest string that you can obtain.

EXPLANATION:

There are a couple of different ways to solve this problem. I'll detail one that converts it to a graph task.

Consider a directed graph G on N vertices, where we add the edge $u \to v$ (where $u < v$) if and only if the character S_u will *always* remain before the character S_v in the final string.
In other words, we add the edge $u \to v$ if and only if we *cannot* (or will not, in case $S_u = S_v$) swap S_u and S_v.

For example, if $S = \text{cbadba}$, we have the edges $1 \to 3, 1 \to 6, 2 \to 4, 2 \to 5, 3 \to 4, 3 \to 6, 4 \to 5, 4 \to 6$.

We only add edges from a lower index to a higher index, so this graph is of course acyclic.

Notice that *any* valid string we can form is obtained from a topological ordering of this graph we created.
Our objective is thus to find the lexicographically smallest topological ordering of G.

Lexicographically smallest toposort

This can be done fairly easily by modifying the standard topological sort algorithm.
Recall that the algorithm repeatedly picks a vertex with indegree zero, and then processes its outedges to update the indegrees of its neighbors.

To obtain the lexicographically smallest ordering, we only need to fix our choice of vertex: instead of choosing an arbitrary zero-indegree vertex, pick the one with lowest value.
This can be done quickly by storing available vertices in a set or heap.

For a graph with E edges and V vertices, the complexity of this is $\mathrm{O}(E + V \log V)$.

We are left with only one problem now: our graph is too large!
There are only N edges, but there can be upto order N^2 edges, which is way too much.

However, notice that many of our edges are redundant.
If we have edges $u \to v$ and $u \to w$ but $S_v = S_w$, then we also have the edge $v \to w$.
This makes the $u \to w$ edge pointless (from the perspective of topological ordering) since $u \to v$ and $v \to w$ together give us the same information anyway.

In particular, this means that for a given index, we only need at most one edge for each character — connect it to the closest occurrence of that character to the right.

This immediately brings the number of edges down to at most $26N$, after which we can easily run our toposort algorithm as outlined above to obtain the answer.

There are other ways to solve this problem as well: for example, it can be solved by building the answer greedily from left to right, but this takes some careful implementation to not TLE.
The setter's code linked below does this.

TIME COMPLEXITY

$\mathrm{O}(26 \cdot N + N \log N)$ per test case

PREREQUISITES:

[Cycle decomposition of permutations]

PROBLEM:

You have an array A of length N.
In one move, you can pick an index i and a non-empty subsequence S of A, and set A_i to the bitwise XOR of the elements in A.

Find a sequence of at most $\lceil \frac{3N}{2} \rceil$ moves that sorts A.

EXPLANATION:

First, let's try to solve a more restricted version of this task: assume that all N elements are distinct (i.e, A is a permutation).

A standard technique when dealing with permutations is to look at their cycle decompositions.
If you're unfamiliar with cycle decompositions of permutations, I recommend reading through the blog linked in the prerequisites.

Let's look at the cycle decomposition of A; in particular, let's look at a specific cycle.

Consider a cycle $C = [c_1, c_2, \ldots, c_k, c_1]$.
This means we need to place value A_{c_i} at position c_{i+1}.

Let's attempt to sort this.
If you try working out a strategy for small cycles on paper (try sorting the arrays $[2, 1], [3, 1, 2], [4, 1, 2, 3]$, for instance), you may find the following:

- Set A_{c_1} to the XOR of the whole cycle
- Set A_{c_2} to the XOR of the whole cycle
- Set A_{c_3} to the XOR of the whole cycle

  $\vdots$
- Set A_{c_k} to the XOR of the whole cycle
- Set A_{c_1} to the XOR of the whole cycle

Why does this work?

Let's look at how the values change after each move.
Note that our values start out at $[A_{c_1}, A_{c_2}, \ldots, A_{c_k}]$ and we want to reach $[A_{c_k}, A_{c_1}, \ldots, A_{c_{k-2}}, A_{c_{k-1}}]$.

Let $X = A_{c_1} \oplus A_{c_2} \oplus \ldots \oplus A_{c_k}$.
Then,

- After the first move, the array is $[X, A_{c_2}, A_{c_3}, \ldots, A_{c_k}]$
- After the second move, the array is $[X, A_{c_1}, A_{c_3}, \ldots, A_{c_k}]$
  - This is because $X \oplus (A_{c_2} \oplus \ldots \oplus A_{c_k}) = (A_{c_1} \oplus A_{c_2} \oplus \ldots \oplus A_{c_k}) \oplus (A_{c_2} \oplus \ldots \oplus A_{c_k}) = A_{c_1}$.
- After the third move, the array is $[X, A_{c_1}, A_{c_2}, A_{c_4}, \ldots, A_{c_k}]$

  $\vdots$
- The array becomes $[X, A_{c_1}, A_{c_2}, \ldots, A_{c_{k-1}}]$
- After the final move, the array becomes $[A_{c_k}, A_{c_1}, A_{c_2}, \ldots, A_{c_{k-1}}]$ which is exactly what we want.

This takes exactly $k + 1$ moves, where k is the length of the cycle.
Note that if the length of the cycle is 1, it corresponds to an element that's already in its place; and hence we

don't require any operations for it.

Now, for each cycle of length k we require $k+1$ operations.
Summing this across several cycles, it can be seen that the total sum is simply the sum of the lengths of the cycles, plus the number of cycles.

The cycles decompose the permutation, so the sum of their lengths is at most N.
Further, there are at most $\lfloor \frac{N}{2} \rfloor$ cycles of length ≥ 2, again because the cycles decompose the permutation.

This bounds the number of operations we make by $N + \lfloor \frac{N}{2} \rfloor$, which is within the limit.

In case A is not a permutation, simply convert it into one!
For each i, let B_i be the pair (A_i, i).
B then consists of distinct elements, and sorting B lexicographically is the same as sorting A so just do that using the above algorithm.

TIME COMPLEXITY

$\mathrm{O}(N^2)$ per test case.

PREREQUISITES:

[Sprague-Grundy and elementary game theory](#) (optional)

PROBLEM:

Chef and Chefina play a game on N piles of stones.
On their turn, a player either removes one stone from a pile, or splits a pile into two strictly smaller piles.

Chef starts first. Under optimal play, who wins?

EXPLANATION:

tl;dr

Let x be the number of piles with an even number of stones, and y be the number of piles with exactly one stone.

Chefina wins if and only if *both* x and y are even.

There are a couple of different ways to come up with a solution to this problem.

Grundy numbers

The 'easiest' way (but also the way that requires some knowledge) is to directly apply the Sprague-Grundy theorem and compute Grundy numbers.

It's clear that each pile is independent, so we can simply compute the grundy number of a single pile, then xor everything together and check whether the final result is non-zero.

Let $g(x)$ denote the grundy number of a pile of x stones.
If you compute grundy numbers for a few small pile sizes using, say, brute-force, you might notice the following pattern:

- $g(0) = 0$
- $g(1) = 1$
- For $x \geq 1$, $g(2x) = 2$ and $g(2x+1) = 0$.

This allows us to compute $g(x)$ for any x in $\mathrm{O}(1)$ time, and the problem is immediately solved.

A direct strategy

Let's first try to solve for a single pile of stones.

- If the pile has 0 stones or 1 stone, the winner is obvious.
- If the pile has a positive even number of stones, Chef can always win by splitting it into two equal-sized piles, and then mirroring Chefina's moves.
- If the pile has an odd number of stones (and ≥ 3), Chef can never win.
  - If Chef removes one stone from the pile, it has an even number remaining so Chefina wins with the previous strategy.
  - If Chef splits the pile, the one of the resulting piles will be odd and the other even.
    - If the odd pile has size ≥ 3, Chefina splits the even pile into 2 equal parts and passes the turn to Chef, who again cannot win: he loses on the smaller odd pile recursively, and he loses on the split even piles since Chefina mirrors his moves there.
    - If the odd pile has size 1 and the even pile has size 2, Chefina makes the piles $(1, 1)$ and hence wins.
    - If the odd pile has size 1 and the even pile has size $x \geq 4$, Chefina splits the even pile into 1 and $x - 1$. Now if Chef takes one 1 Chefina takes the other; and otherwise Chef has to play on a smaller odd pile which is again losing.

This tells us that on a single pile, Chef wins if it's either a single stone or even-sized.
Further, Chefina's winning strategy often relies on mirroring Chef's moves.

Let's attempt to generalize this to multiple piles.

- If there's an even number of 1-stone piles, Chef can win half of them and Chefina wins the other half, so it's in Chefina's favor since she'll make the last move.
- If there's an even number of even-sized piles, once again they win half each so it's in Chefina's favor.

In particular, if there's an even number of 1-sized piles *and* an even number of even-sized piles, Chefina will always win.

What about the other cases?
It turns out that Chef can always win them!

Proof

Suppose there are x piles of size 1 and y of even size.
If $(x+y)$ is odd, it's obvious that Chef can always win since he has strictly more winning piles.

Now, suppose x and y are both odd.
There's at least one even-sized pile, so take it; let it have K stones.

- If $K = 2$, remove one stone from it. This increases x by 1 and decreases y by 1, so they're both even. It's Chefina's turn, so she loses.
- If $K \geq 4$, split into two piles of sizes $(1, K-1)$. Again, this increases x by 1 and decreases y by 1, so once again x and y both become even and Chefina loses.

TIME COMPLEXITY

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Binary search

PROBLEM:

$F(N)$ denotes the difference between the number of squares and the number of cubes from 1 to N.

Given X, find the smallest value of N such that $F(N) \geq X$.

EXPLANATION:

Problems of the type "find the smallest N such that …" are often solved with binary search.
Unfortunately, that requires the function to be monotonic, and $F(N)$ here isn't monotonic.

Analyzing how it changes, we can see that:

$$F(N) = \begin{cases} F(N-1), & \text{if } N \text{ is neither a square nor a cube} \\ F(N-1)+1, & \text{if } N \text{ is a square but not a cube} \\ F(N-1)-1, & \text{if } N \text{ is a cube but not a square} \\ F(N-1), & \text{if } N \text{ is both a square and a cube} \end{cases}$$

In particular, notice that F can only increase at a square number, so the final answer must be a square number.
Further, the fact that cubes are more 'spread out' than squares means that F is actually a monotonic function when evaluated on the squares!
That is, we consider the function $g(N) = F(N^2)$ instead, which is increasing.

Proof

It's enough to prove that there's at most one cube lying between any two consecutive squares, since this is what makes F increasing on the squares.

So, suppose we have $1 \leq x^2 \leq y^3 \leq (y+1)^3$. We just need to prove $(x+1)^2 < (y+1)^3$.

This is not hard to do by simple algebraic manipulation.
Expand out $(x+1)^2$ and $(y+1)^3$, to obtain (x^2+2x+1) and $(y^3+3y+3y^2+1)$.

Comparing their terms, we can see that:

- $y^3 \geq x_2$
- They both have a 1
- $3y^2 \geq 3x > 2x$. This follows from the fact that $x^2 \leq y^3$ means that $y^2 \geq x^{4/3} \geq x$.

So we have the required inequality.

With this information in hand, the solution is simple: use binary search to find the smallest value of N such that $g(N) \geq X$; the final answer is then N^2.

This requires us to quickly evaluate $g(N) = F(N^2)$.

Note that the number of squares that are $\leq N^2$ is simply N, so we just need to count the number of cubes that are $\leq N^2$.
There are several ways to do this: for example, use `std::cbrtl` or yet another binary search.

Regarding precision

You may have encountered precision errors when computing cube roots; luckily, one of the samples was specifically designed to catch several common implementations.

In particular, `cbrt` in C++ or something like `N ** (1 / 3)` in Python have precision issues.

There are several ways to get around this, some being problem dependent:

- If possible, the absolute best solution is to simply not deal with floating-point integers at all. In this problem, the number of cubes $\leq N$ can be computed using a binary search across the integers, so this is the safest way.
- `cbrtl` in C++ is precise enough to get AC in this task.
- The cube root can be manually adjusted once you compute it: if c is the value you compute as the cube root of N (after rounding) and $(c+1)^3 \leq N$, then increase c to $c+1$ instead.
- After computing the cube root, add a small value to it (like 10^{-10}) before rounding to an integer.

TIME COMPLEXITY

$\mathrm{O}(\log^2 X)$ per testcase.

PREREQUISITES:

Breadth-first search

PROBLEM:

You are given an $N \times N$ grid, each cell of which contains either 0 or X.
You can move from cell (x_1, y_1) to (x_2, y_2) if they both contain 0, and lie in the same row or column.

Answer Q independent queries of the following form:
Given (x, y), set cell (x, y) in the grid to contain X. Then, find the shortest path that starts in the first row, ends in the first row, contains distinct cells, and moves outside the first row at least once.

EXPLANATION:

Ignore the queries for now: let's see how to compute the answer for a fixed grid.

Of course, the obvious method that comes to mind is to construct the underlying graph and attempt to compute the required quantity in it, perhaps by performing multiple BFS-es.

However, the graph we construct has $\mathrm{O}(N^2)$ vertices and $\mathrm{O}(N^3)$ edges, so performing a BFS $\mathrm{O}(N)$ times on it is infeasible; and certainly it seems impossible to deal with updates to the grid.

Let's construct a different graph model instead; in fact, one that is a rather standard idea on grid graphs.

Consider a bipartite graph with $2N$ vertices, with an edge between i and $N + j$ if and only if $G_{x,y}$ is 0.
Essentially, $1, 2, 3, \ldots, N$ represent the rows of the grid, and $N + 1, \ldots, 2N$ represent the columns.
Let's name this graph Γ.

Note that any path in the grid corresponds to a path of *edges* in this graph.

Now, let's look at the shortest valid path in the grid.
It's always optimal for the first move to be downwards from the first row, i.e, an edge of the form $(1, x)$ for some $x > N$.
Similarly, it's always optimal for the last move to be upwards from outside the first row, i.e, another edge of the form $(1, y)$ for $y > N$. Note that $y \neq x$ must also hold since the cells on the path are distinct.

Further, an optimal path will never make two horizontal (or two vertical) moves in a row, since they can be replaced with a single move.
In other words, our sequence of moves looks like vertical $\rightarrow$ horizontal $\rightarrow$ vertical $\rightarrow \ldots \rightarrow$ vertical.

Notice that this simply corresponds to a cycle in Γ that starts and ends at vertex 1; and in fact, the number of vertices in the path equals the length of the cycle.

So, our aim is to simply compute the shortest cycle passing through vertex 1 in Γ.

Finding the shortest cycle through a vertex

Let's fix a vertex u, and say we want to find the shortest cycle that passes through u in a given graph.

Finding this can be done with BFS in $\mathrm{O}(|V| + |E|)$, as follows:

First, run a BFS from vertex u and compute a BFS tree.
Now, the shortest cycle through u will look like $u \rightarrow \ldots \rightarrow x \rightarrow y \rightarrow \ldots \rightarrow u$, where x and y satisfy the following conditions:

- There is an edge $x \rightarrow y$ that is **not** a tree edge

- The lowest common ancestor of x and y in the BFS tree is u

The length of such a cycle is, of course, $dist(u, x) + dist(u, y) + 1$, where the distances can be computed during the BFS.

So, simply check the second condition for every non-tree edge and take the minimum cycle length across all of them that satisfy it.
Note that you don't need to actually compute the LCA of any pair: the only check that needs to be done is whether x and y lie in the subtrees of different children of u, so for each vertex simply maintain which child's subtree it lies in.

In the case of our Γ, we have $2N$ vertices and $\mathrm{O}(N^2)$ edges, so this process takes $\mathrm{O}(N^2)$ time.

Dealing with updates

Each update sets one `0` in the grid to `X`.
Notice that in terms of Γ, this simply correponds to deleting an edge!

Once we find the shortest cycle in Γ containing 1, note that it has $\leq 2N$ edges.
If the deleted edge isn't among them, the answer doesn't change, since deleting an edge can't give us a shorter cycle.

Then, for each edge on the shortest cycle, delete it and once again compute the answer for the resulting graph.

This gives us an algorithm in $\mathrm{O}(N^3)$ in total, after which each query is answered in $\mathrm{O}(1)$ time.

TIME COMPLEXITY:

$\mathrm{O}(N^3 + Q)$ per testcase.

PROBLEM:

Two players (Zlatan and Ramos) play a game on a binary string of length N.
On their turn, a player chooses two adjacent unequal characters from the string and deletes them both.
Under optimal play, who wins?

EXPLANATION:

The string is a binary string, and each move deletes an adjacent pair of unequal characters.
So, each move will delete exactly one 1 and one 0 from the string.

Further, if the string contains at least one 1 and at least one 0, then there will definitely exist some pair of adjacent unequal characters, so a move can always be made.
Turning this around, the only time a move cannot be made is when the string consists of only a single character.

So, suppose S contains c_0 zeros and c_1 ones.

Each move decreases c_0 and c_1 by one each. This means that $\min(c_0, c_1)$ moves can certainly be made, no matter what those moves are.
On the other hand, after $\min(c_0, c_1)$ moves, one of the characters will be fully exhausted and no more moves can be made, so the game always lasts exactly $\min(c_0, c_1)$ moves!

This means that the first player (Zlatan) wins if $\min(c_0, c_1)$ is odd, and loses if it's even.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

You're given a permutation of $\{1, 2, \ldots, N\}$.
In one move, you can:

- Swap A_i and A_j, where $|i - j| > 1$.
  This costs one coin.
- Pick indices $i < j < k$ such that $A_i > A_j < A_k$, and delete A_i from the array.
  This costs zero coins.

Find a sequence of operations that reaches a sorted array using the least number of coins possible, or claim that it's impossible to reach a sorted array.

EXPLANATION:

The absolute best case is when we need to use 0 coins, i.e, only the delete operation is allowed.
Let's try to figure out when this is possible.

Note that since we must choose $i < j < k$ and delete A_i, in particular the last two elements of A will never be deleted.
So, if the final array is to be sorted, certainly $A_{N-1} < A_N$ must hold.
In fact, this condition is also sufficient: if $A_{N-1} < A_N$, there always exists a sequence of deletions resulting in a sorted array.

Proof with construction

Let's fix $k = N$ as a constant.
Let $j = N - 1$ initially. We'll maintain $A_j < A_k$ at all times.

Then, for each index i from $N - 2$ down to 1:

- If $A_i > A_j$, then we know $A_i > A_j < A_k$. So, delete A_i from the array.
- If $A_i < A_j$, set $j = i$ and continue on.

Notice that after each step, the non-deleted elements from [A_i, A_{i+1}, \ldots, A_N]$ form a sorted sequence (we only add an element to the start of the sequence, and only when it's smaller than the previous first element).
So, when the process ends, we'll have a sorted array and we're done!

This gives us a hint as to how to proceed: if we end up with $A_{N-1} < A_N$, we'll be done.

Intuitively, this means the minimum number of coins needed is also pretty small: for example, A_{N-1} and A_N can be swapped using three operations (fix some $i \le N - 3$, then $swap(A_i, A_{N-1}), swap(A_i, A_N), swap(A_i, A_{N-1})$).
So, the answer is surely ≤ 3 coins.

Checking if 0 coins are needed is as mentioned above.

Checking if 1 coin is needed is also trivial: for each i from 1 to $N - 2$, check whether swapping A_i with A_{N-1} or A_N will allow for $A_{N-1} < A_N$; if yes, perform this swap and then apply the 0 coin algorithm.
Note that A_{N-2} can't be swapped with A_{N-1} because of the distance condition, make sure to take care of that.

We check 2 swaps for each index, and do non-trivial work at most once, so it's still $\mathrm{O}(N)$.

If one swap isn't enough, then two swaps will always be enough (or, in a small handful of cases, the answer is `No`).

This can be done as follows:

- If $N \geq 5$, then A_1 and A_2 can always be swapped with A_N and A_{N-1}.
  So, swap $\min(A_1, A_2)$ with A_{N-1} and the other one with A_N.
  Now $A_{N-1} < A_N$.
  In fact, this even works for $N = 4$ when $A_1 < A_2$.
- This leaves us with $3 \leq N \leq 4$ to deal with. You can either bruteforce all possible swaps/deletes here, or do a bit of casework:
  - If $N = 3$ and it hasn't been sorted yet, notice that $A_2 = 3$ will be the case.
    In such cases, the answer is `No`.
  - If $N = 4$ and it hasn't been sorted yet, the only possible arrays are $[3, 1, 4, 2]$ and $[3, 2, 4, 1]$.
    These can both be solved using two swaps: $swap(A_1, A_4)$ followed by $swap(A_1, A_3)$ makes $A_3 < A_4$.

The problem is now solved.

You might notice that we performed all swap operations before any delete operations, though we're allowed to interweave them as we please.
However, performing deletes after swaps doesn't improve optimality anyway.

Proof

If 0 coins are needed by our algorithm, no swaps are performed and it's surely optimal.

Otherwise, we have $A_{N-1} > A_N$.
This relation will not change no matter how many deletions are done; which means that at least one swap is definitely needed.
So, the minimum number of coins needed is ≥ 1.
In particular, that means our solution for the case when exactly one coin is used is also optimal.

Now, let's look at when we used two coins.
We have $A_{N-1} > A_N$.
Further, a single swap wasn't enough to reverse this relation. That means:

- $A_{N-1} = N$ (otherwise N could be placed at A_N)
- $A_N < A_i$ for every $1 \leq i \leq N-3$; otherwise swapping A_i and A_{N-1} would satisfy the condition.
  Notice that since A is a permutation, this means A_N *must* be either 1 or 2.
  Further, if $A_N = 2$, then $A_{N-2} = 1$ will hold (otherwise 1 could be swapped to position $N-1$).

The final observation allows us to prove optimality via induction on N.

That is, consider a permutation of $\{1, 2, \ldots, N\}$ (where $N \geq 4$) that's of either the form $[\ldots, N, 1]$ or $[\ldots, 1, N, 2]$.
We claim that any such permutation requires at least 2 coins to be solved.
This can be verified manually for $N = 4$, so let's look at $N > 4$.

- If the first move is a deletion, we end up at the same pattern but with $N-1$ — this is because the last two elements can't be deleted; and in the case $[\ldots, 1, N, 2]$ the 1 can't be deleted either.
  By inductive hypothesis we still require 2 coins.
- if the first move is a swap, then we know for sure that $A_{N-1} > A_N$ still holds (because no single swap can overcome this), and so at least one more coin is needed.
  This means ≥ 2 coins are needed in total, and we're done!

TIME COMPLEXITY

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Basic combinatorics

PROBLEM:

Given a permutation P of length N, find the number of non-empty subsequences of P that have the same number of inversions as it.

EXPLANATION:

For a subsequence S of P, let $inv(S)$ denote its inversion count.
It should be clear that for any subsequence S, $inv(S) \le inv(P)$ must hold.
So, let's try to figure out when $inv(S) < inv(P)$ holds.

Consider some inversion (i,j) in P, i.e, $i < j$ and $P_i > P_j$.
If a subsequence S doesn't contain *both* i and j, then this inversion won't be counted in $inv(S)$, and hence $inv(S) < inv(P)$ will hold.

Turning this around, for $inv(S) = inv(P)$ to be true, S should contain *every* index that's involved with at least one inversion.
It's not hard to see that this condition is both necessary and sufficient, i.e, $inv(S) = inv(P)$ if and only if S contains every index that's part of an inversion.
So, our task shifts to computing what these positions are.

An index i is part of an inversion iff:

- There exists an index $j < i$ such that $P_j > P_i$; or
- There exists an index $j > i$ such that $P_i > P_j$

That means that index i is *not* part of any inversion iff $P_i > P_j$ for every $j < i$, *and* $P_i < P_j$ for every $j > i$.
In other words, $P_i = \max(P_1, P_2, \ldots, P_i)$ and $P_i = \min(P_i, P_{i+1}, \ldots, P_N)$.

By precomputing prefix maximums and suffix minimums, this can be checked for each index in $\mathrm{O}(1)$.

Now, we know exactly which indices *must* be included in any valid subsequence S.
Each of the other indices may be included or not; they don't affect the inversion count.

So, if there are k 'free' indices, the number of subsequences is 2^k.
Note that there's a single edge-case here: if $k = N$, then the answer is $2^k - 1$ instead, since the empty sequence will be counted in 2^k and we don't want that.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PROBLEM:

Given X and Y, find any positive integer N such that Y divides $N+X$ and X divides $N+Y$.

EXPLANATION:

Let's analyze what a valid N must look like.

- Y should divide $N+X$, which means $N+X=aY$ for some *positive* integer a.
- X should divide $N+Y$, which means $N+Y=bX$ for some *positive* integer b.

Putting these equations together, we see that $N=aY-X=bX-Y$, which reduces to

$$X\cdot(b+1)=Y\cdot(a+1)$$

Our task is to choose appropriate a and b so that these equations are satisfied.

The simplest way to do this is to make both sides equal to XY, by choosing $b=Y-1$ and $a=X-1$.
This gives us $N=aY-X=(X-1)Y-X=XY-X-Y$.

However, note that we need to find a positive value of N, and the above value isn't always positive.
In fact, it fails to be positive in exactly three cases: $X=1$, $Y=1$, or $X=Y=2$.

For those three cases, we simply need to choose higher values of a and b.
For example, choose $a=3X-1$ and $b=3Y-1$ to obtain $N=3XY-X-Y$.

TIME COMPLEXITY:

$\mathrm{O}(1)$ per testcase.

PREREQUISITES:

Combinatorics

PROBLEM:

The beauty of an array B equals $\sum_{i=1}^{M} |B_i - B_{M+1-i}|$.
The score of array B equals the maximum beauty across all its permutations.

Given N and X, find the sum of scores of all arrays of length N whose elements lie between 1 and X.

EXPLANATION:

First, let's figure out what the score of an array is.

Notice that index i and index $M + 1 - i$ are symmetric about the center of the array, so indices are essentially paired up. The difference of each such pair is counted twice.
Consider some index i, with $B_i = x$ and $B_{M+1-i} = y$.
Then, these indices contribute $2 \cdot |x - y| = 2\max(x, y) - 2\min(x, y)$ to the beauty of the array.

Since indices are paired up, from a global perspective this means that exactly $\lfloor \frac{N}{2} \rfloor$ indices contribute a *positive* value to the beauty, and exactly $\lfloor \frac{N}{2} \rfloor$ indices contribute a *negative* value.

To maximize this, clearly the best we can do is to ensure that the largest $\lfloor \frac{N}{2} \rfloor$ values contribute positively, and the smallest $\lfloor \frac{N}{2} \rfloor$ values contribute negatively.
This is of course easily achievable: simply sort the array!

The above discussion tells us exactly what the score of a fixed array B is — it's twice the sum of the largest $\lfloor \frac{N}{2} \rfloor$ elements, minus twice the sum of the smallest $\lfloor \frac{N}{2} \rfloor$ elements.
Let's use this information to solve the problem.

Suppose we were able to calculate $\text{ct}[i][y]$, the number of arrays whose i-th smallest element is y.
This would allow us to solve the problem quite easily: after all, the answer would either increase or decrease by $2 \cdot y \cdot \text{ct}[i][y]$, depending on the value of i.

However, directly computing $\text{ct}[i][y]$ is not easy.

Instead, let's try to relax the problem a little.
Let $g[i][y]$ be the number of arrays whose i-th smallest element is $\leq y$.
That is, $g[i][y]$ is the number of arrays containing *at least* i elements that are $\leq y$.
Then, $\text{ct}[i][y] = g[i][y] - g[i][y-1]$, so if we're able to compute g we can easily compute ct.

However, once again it's not clear how we would compute g. This time, there's too much freedom: sure, we can place i elements $\leq y$ in the array, but there might be other elements $\leq y$ as well. How to account for those?

Well, we can try adding that as a constraint.
Let $h[i][y]$ denote the number of arrays whose i-th smallest element is $\leq y$, and whose $(i+1)$-th element is $> y$.
In other words, $h[i][y]$ is the number of arrays such that *exactly* i elements are $\leq y$.

$h[i][y]$ is in fact quite easy to compute:

- Fix the positions of the i elements that are $\leq y$. This can be done in $\binom{N}{i}$ ways.
- Each of these i positions has y choices, since we can place any of $1, 2, 3, \ldots, y$ there.
  This is y^i ways.

- Each of the other positions has $X - y$ choices, since we can place any of $y + 1, y + 2, \ldots, X$ there. This is $(X - y)^{N-i}$ ways.

So, $h[i][y] = \binom{N}{i} \cdot y^i \cdot (X - y)^{N-i}$, which can be computed in $\mathrm{O}(\log N)$ or even $\mathrm{O}(1)$ with some precalculation.

Since $h[i][y]$ is the number of arrays with exactly i elements $\leq y$, and $g[i][y]$ is the number of arrays with at least i elements $\leq y$, we have

$$g[i][y] = h[i][y] + h[i+1][y] + \ldots + h[N][y]$$

This is basically a (column-wise) suffix sum of the h array, so is easy to compute once h is known.

So,

- Compute all the $h[i][y]$ values, which can be done in $\mathrm{O}(N \cdot X)$.
- Take column-wise suffix sums of h to obtain the $g[i][y]$ values, which once again takes $\mathrm{O}(N \cdot X)$ time.
- From the $g[i][y]$ values, compute $\mathrm{ct}[i][y]$ and hence solve the problem.

TIME COMPLEXITY

$\mathrm{O}(N \cdot X)$ per test case.

Why does this work?

Let's look at even and odd indices separately.

For an even index, say $2i$, the corresponding sum is $1 + 2 + \ldots + 2i = \frac{2i(2i+1)}{2} = i(2i + 1)$. This is not divisible by $2i$.

For an odd index, say $2i + 1$, the corresponding sum is $1 + 2 + \ldots + (2i - 1) + 2i + (2i + 2) = i(2i + 1) + (2i + 2)$.
This can be further simplified into $(2i + 1)(i + 1) + 1$, which isn't divisible by $(2i + 1)$ (since it leaves a remainder of 1 when divided by $2i + 1$).

This way, both the odd and the even indices are satisfied, so we're done.

TIME COMPLEXITY

$\mathrm{O}(N)$ per test case.

PREREQUISITES:

Segment trees with lazy propagation

PROBLEM:

You have an array A and an integer K.
Let $f(l, r)$ denote the number of distinct integers in the subarray $[A_l, A_{l+1}, \ldots, A_r]$.

Find

$$\sum_{l=1}^{N} \sum_{r=l}^{N} f^k(l, r)$$

EXPLANATION:

First, let's try to solve this problem for $K = 1$, and then generalize that solution to higher K.

Solving $K = 1$

Our task is to compute the sum of the number of distinct elements across all subarrays.
While counting the number of distinct elements in a range is easy in $\mathrm{O}(N)$, and even answering Q queries about different ranges is doable in $\mathrm{O}((N + Q) \log N)$ (see **here** for several ways to do it, both online and offline), querying for every subarray is still too slow.

However, we can slightly modify one of the ways to answer queries offline to achieve what we want.

Let's fix r, and then let D be an array of size N such that $D_l = f(l, r)$.
If we know D for r, we can quickly update it for $r + 1$ using the following observations:

- If A_{r+1} occurs in the subarray $A[l, r]$, then $f(l, r + 1) = f(l, r)$; i.e, there's no need to change D_l.
- Otherwise, $f(l, r + 1) = f(l, r) + 1$; i.e, D_l increases by exactly 1.
- Further, notice that if D_l remains unchanged, so does D_{l-1}; and if D_l increases by 1, so does D_{l+1}.

Essentially, this induces a range update on D: a certain range of values is increased by 1, and everything else remains unchanged.
Finding which range increases isn't too hard either: we just need to find the previous occurrence of A_{r+1}.

In particular, we have the following algorithm:

- Let D be as described above, initially containing all zeros.
- We'll consider r from 1 to N.
- At r, let p be the position of the previous occurrence of A_r.
- Then, add 1 to the range $[p + 1, r]$ of D.
- Finally, add $D_1 + D_2 + \ldots + D_r$ to the answer.

To do this quickly, we need a data structure that allows us to quickly add a value to a range, and find the sum of values in a range.
This can be done using a segment tree with lazy propagation.

We do N updates and N queries, so $K = 1$ is solved in $\mathrm{O}(N \log N)$.

Solving $K > 1$

Now, we need to generalize this solution to higher K.

Let's do almost exactly the same thing; except that now, $D_l = f^K(l, r)$.

Our updates and queries are also exactly the same; the only difference is in the part where we propagate updates down the segment tree.
In particular, we need to update the sum of K-th powers of several values, if all those values are increased by x.

Suppose we know $\sum_i a_i^K$, and we want to compute $\sum_i (a_i + x)^K$. That can be done as follows:

$$\begin{aligned}\sum_i (a_i + x)^K &= \sum_i \sum_{j=0}^{K} \binom{K}{j} a_i^j x^{K-j} \\ &= \sum_{j=0}^{K} \binom{K}{j} x^{K-j} \sum_i a_i^j\end{aligned}$$

Notice that the second summation is simply the sum of j-th powers of all the a_i, and is information that doesn't depend on x.
If we maintained that information, this update can be done in $\mathrm{O}(K)$.

So, in each node of the segtree we keep the sum of the j-th powers of the values in it, for each $0 \le j \le K$.
When pushing an update, simply update all of these $K + 1$ sums as mentioned above: this takes $\mathrm{O}(K^2)$ time — $\mathrm{O}(K)$ for each.

This gives us a solution in $\mathrm{O}(K^2 N \log N)$.

TIME COMPLEXITY:

$\mathrm{O}(K^2 N \log N)$ per testcase.

PREREQUISITES:

Combinatorics, [the inclusion-exclusion principle](), [polynomial convolution with arbitrary mod]()

PROBLEM:

A permutation P of length $2N$ is called N-beautiful if the N pairs $(P_i \bmod N, P_{i+N} \bmod N)$ are all distinct.

Given K, For each $1 \le i \le K$ find the number of i-beautiful permutations, modulo a given prime number M.

EXPLANATION:

We'll first solve subtask 1, and then extend that solution to subtask 2.

Subtask 1

The limit on K is small enough here that $\mathrm{O}(K^2)$ will pass.
In particular, we can solve for a fixed N in $\mathrm{O}(N)$ time; so let's attempt to do that.

Note that we only really care about the values of $P_i \bmod N$, which range from 0 to $N-1$. Also, we have exactly two of each value.
We'll call arrays of length $2N$ containing each integer from 0 to $N-1$ twice *good arrays.*

Let's call a permutation that isn't N-beautiful, N-ugly.
We'll try to count the number of N-ugly permutations.

Suppose we have a N-ugly permutation.
Then, there exist two indices $1 \le i < j \le N$ such that $P_i = P_j$ and $P_{i+N} = P_{j+N}$.
Such a pair will be called a *bad pair*.

In particular, note that these four positions use all copies of P_i and P_{i+N}, so the rest of the elements are simply a good array of length $2N-4$ (after appropriate relabelling).

It's not too hard to count the number of such arrays, since fixing i, j, the values P_i and P_{i+N}, and which copy of each value goes to which position is simple math and a bunch of multiplications.

However, we can't directly subtract this count from $(2N)!$, since a good array with multiple bad pairs was counted multiple times!
This should lead you to the idea of using the inclusion-exclusion principle.

Let B_i be the number of good arrays with *at least* i bad pairs.
By the principle of inclusion-exclusion, the answer we're looking for is

$$\sum_{i=0}^{N} (-1)^N B_i$$

So, let's focus on computing B_i quickly.
This is a generalization of how we computed B_1, and is just a bunch of combinatorics.

Details

There are i bad pairs, which fixes $2i$ positions among the first N. These can be chosen in $\binom{N}{2i}$ ways.
We need to now split these $2i$ positions into i pairs.
This can be done in $\binom{2i}{2,2,2,\ldots,2} = \frac{(2i)!}{2^i}$ ways by definition of the multinomial coefficient.

Next, we need to fix which elements go into each pair. Let's fix an order of the pairs and assign values one by one.
This can be done in $N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot (N-2i+1)$ ways, which after a little algebraic manipulation is seen to be just $\binom{N}{2i}(2i)!$.

However, we also need to divide by $i!$ since the order in which we assign values to the pairs doesn't actually matter (it makes no difference to the final permutation).

Each bad pair now has four choices (we can swap elements with the same value modulo N to obtain a different permutation with the same bad pair count).
This gives us 4^i choices in total.

Finally, there are $(2N-4i)!$ ways to arrange the elements not among these bad pairs.

Putting everything together, we have

$$B_i = \binom{N}{2i}\binom{N}{2i}\frac{4^i(2i)!(2i)!(2N-4i)!}{2^i i!}$$

which can be computed in $\mathrm{O}(\log N)$ or even $\mathrm{O}(1)$ time, since everything is either a factorial or a power of two (or an inverse of one of them)

Since each B_i can be computed quickly, we can solve for a fixed N in $\mathrm{O}(N)$ time, which is good enough for subtask 1.

Subtask 2

Let's look at our expression for B_i again, and clean it up a bit by cancelling out common terms.
We obtain

$$B_i = \frac{2^i N!N!(2N-4i)!}{(N-2i)!(N-2i)!i!} = \frac{2^i N!N!}{i!}\binom{2N-4i}{N-2i}$$

The $N!$ terms are independent of i, so let's ignore them for now.
Notice that we have the product of an "i-part" comprising $\frac{2^i}{i!}$ and something like an "$N-i$ part" comprising $\binom{2N-4i}{N-2i}$.

This should make you think along the lines of convolution.
However, we can't directly multiply two polynomials: the $N-i$ part doesn't quite depend on $N-i$, it depends on $N-2i$ (and there's no apparent nice way to make the i-part depend on $2i$ instead).
It'd be nice if we had $2N$ instead of N, so let's do just that!

We split the computation into two cases: even N and odd N. Let's deal with the even case first.
So, suppose $N = 2n$.

We now have $\binom{2N-4i}{N-2i} = \binom{4n-4i}{2n-2i} = \binom{4(n-i)}{2(n-i)}$, which does depend on $n-i$.

So, create the polynomials p and q, with coefficients

- $p_i = \frac{(-1)^i 2^i}{i!}$
- $q_i = \binom{4i}{2i}$

and multiply them together so obtain r.

The answer for $2n$ is then r_n, multiplied by $(2n)!^2$ (which we took out at the start).

For odd N, a similar technique works: write $N = 2n + 1$, and once again you'll get a very similar relation on $n - i$. Following this, multiply the appropriate polynomials and we're done.

Note that this problem requires multiplying two polynomials and keeping their coefficients modulo some M, where M isn't necessarily 'nice'.
Standard FFT and NTT won't suffice here; a tutorial on how to do this can be found at **cp-algorithms**.

TIME COMPLEXITY

$\mathrm{O}(K \log K)$ per testcase.

PROBLEM:

You have three numbers A, B, C. In one move, you can increase two of them by 1 and decrease the third by 1. Find the minimum number of moves needed to make all three equal, or decide that it's impossible.

EXPLANATION:

First, notice that the given operation always flips the parity of *all* three numbers since each one changes by $+1$ or -1.
If they're to be equal in the end, all three will have the same parity.

So, if all three don't have the same parity initially, it's immediately possible to say that making them equal is impossible.

Now let's look at what we can do with our moves.

Notice that any move increases the sum $(A + B + C)$ by exactly 1.
In particular, this means if we make them all equal to x in the end, the number of moves used will be exactly $3x - A - B - C$ (that is, the difference between the final sum and the initial sum).

Minimizing this is thus the same as minimizing x, so let's attempt to do that.

Suppose $A \le B \le C$.
It's not hard to see that if we make everything equal to x, then $x \ge \frac{B+C}{2}$

Why?

Let d be the *initial* value of $B + C$.

Look at the value of $B + C$ as we perform operations.
Any move does one of two things:

- Increases $B + C$ by 2; or
- Doesn't change $B + C$.

Either way, the value of $B + C$ cannot decrease as we perform operations.
In particular, if we end up with $B = C = x$, then $x + x = 2x \ge d$, which proves our claim.

In fact, we can always achieve $x = \frac{B+C}{2}$
Here's how:

- Let $k = \frac{B+C}{2}$ be the target value.
- Recall that B and C have the same parity, so let $C - B = 2y$ where $y \ge 0$.
- Perform y moves that decrease C and increase the other two.
  - Now we have $B = C = k$, and $A \le k$
- Next, alternate moves that decrease C with moves that decrease B. This way, after two moves A will increase by 2 while B and C will remain unchanged.
- The parity condition ensures that A will reach k after an even number of moves, and we're done!

So, when $A \le B \le C$ and the parity condition is satisfied, the answer is simply

$$3 \cdot \frac{B + C}{2} - A - B - C$$

TIME COMPLEXITY

$O(1)$ per test case.

PREREQUISITES:

Segment trees/Fenwick trees

PROBLEM:

Alice, Bob, and Charles participated in N contests; and obtained A_i, B_i, C_i points in the i-th one, respectively.

A person gets *first place* in a range $[L, R]$ of these contests if their total score in this range is not less than the score of the other two.
For each person, find the longest range of contests in which they obtained first place.

EXPLANATION:

Let's solve the problem for Alice; the same process can be repeated for Bob and Charlie.

Consider some range $[L, R]$ of contests.
Alice's score is not worse than Bob's in this range if and only if $A_L + A_{L+1} + \ldots + A_R \geq B_L + B_{L+1} + \ldots + B_R$.
This can be rewritten as $(A_L - B_L) + (A_{L+1} - B_{L+1}) + \ldots + (A_R - B_R) \geq 0$.

So, if we create an array X with $X_i = A_i - B_i$, we want $X_L + X_{L+1} + \ldots + X_R \geq 0$.
Subarray sums naturally lend themselves to prefix sums, so let's look at those.
If pX denotes the prefix sum array of X, then we want $pX_R - pX_{L-1} \geq 0$, i.e, $pX_R \geq pX_{L-1}$.

Similarly, we can construct arrays Y and pY corresponding to $A_i - C_i$.

Our task is to thus find the longest range $[L, R]$ such that $pX_R \geq pX_{L-1}$ **and** $pY_R \geq pY_{L-1}$.

In particular, if we fix a value of R, we then want to find the smallest $L \leq R$ such that these conditions hold.
There are ways to do this directly using persistent of 2D data structures, but there's also a simpler solution using more well-known (and easy-to-code!) structures.

Let's iterate over values of R in increasing order of pX_R.
This ensures that the condition $pX_R \geq pX_{L-1}$ is automatically satisfied, so we only need to deal with $L \leq R$ and $pY_R \geq pY_{L-1}$.

We now want to find the smallest value of $L \leq R$ such that $pY_R \geq pY_{L-1}$.
That can be done using a segment tree and binary search, as follows:

- Consider a range-minimum segment tree built on the array pY.
- With the help of this tree, we can use binary search to find the first L in the range $[1, R]$ whose pY_{L-1} value is no larger than pY_R.
  - This can be done by querying for the minimum value in the suffix $[1, \text{mid}]$ of the binary search and checking whether it's below pY_R or not.
- Since we're iterating over R in increasing order of pX_R, we can simulate this in the segment tree by first initializing everything to ∞ (some large number) so that querying positions with larger pX-value doesn't affect the minimum.
- After fixing R and finding the optimal L, update position R into the segment tree so that it can contribute to future queries.

At each index, we do a binary search which has segment tree queries within it.
This leads to a complexity of $\mathrm{O}(N \log^2 N)$.

This can be further optimized to $\mathrm{O}(N \log N)$ by combining the binary search into the segment tree query; in the technique sometimes known as **segment tree descent or walk**.
This is not necessary to get AC, however.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PROBLEM:

You're given an integer represented by a binary string of length N.
Can it be represented as the sum of three powers of two?

EXPLANATION:

First off, notice that N is quite large: so large that it's impossible to directly convert it to an integer reasonably (in C/C++, at least).
This should hint towards the fact that we just need to work with the binary representation given to us.

The main observation that needs to be made here is that, for any $m > 0$, we have $2^m = 2^{m-1} + 2^{m-1}$.
That is, we can split a single power of 2 into two powers of 2.
However, there's no way to combine *distinct* powers of 2 into one; so essentially, we can only increase the number of powers of 2.

Now, suppose the string has k ones. This means the given integer is written as the sum of k distinct powers of 2; and by the property of the binary system, this k is *minimal*.
Let's look at a few cases:

- If $k > 3$, then the answer is `No`; we have too many powers of 2 and can't reduce them.
- If $k = 3$, then the answer is clearly `Yes`; since the number is the sum of three distinct powers of 2.
- If $k = 2$, then the answer is still `Yes`; split the larger power of 2 into two smaller ones.
- If $k = 1$, then the answer is `Yes` sometimes. Notice that we need to split this power of 2 into three, so we need enough 'space' to do this (since the exponent will reduce by 2).
  - So, if $k = 1$ and the length of the string is ≥ 3, the answer is `Yes`.
  - Otherwise, the answer is `No`.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PROBLEM:

Given the blood types of N people, and knowledge of which blood type can donate to which blood type, find the maximum number of people in a blood transfusion chain.

EXPLANATION:

Looking at the possible donors/receivers, we see that:

- Blood type `O` can donate to any type.
- Blood type `AB` can receive from any type.
- Blood types `A` and `B` don't have any interaction with each other.

This means:

- Anyone with blood type `O` or `AB` can always be placed in any chain: either at the front, or at the end depending on their type.
- The chain can't contain both a blood type-`A` person and a type-`B` person.

Together, this tells us that we can pick everyone with types `O` and `AB`; and after that we must choose between types `A` and `B`. Clearly, it's best to choose whichever type has most people.

So, if c_x denotes the number of people with blood type x, the answer is

$$c_O + c_{AB} + \max(c_A, c_B)$$

which can also be written as

$$N - \min(c_A, c_B)$$

since we're essentially excluding the people of either `A` or `B` type and taking everyone else.

The counts of the four types can be found by iterating across the array.

TIME COMPLEXITY

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

DFS

PROBLEM:

There's a tree on N vertices. Alice and Bob start at vertices A and B of this tree and play a game.
The i-th vertex has a population of P_i.

Each turn proceeds as follows:

- If either Alice or Bob cannot move, the game ends.
- Otherwise, Alice moves to an unvisited (by her) vertex, and then Bob moves to an unvisited (by him) vertex.
- Alice receives a point if the total population of the vertices visited by her so far exceeds the total population of the vertices visited by Bob.

Alice moves to maximize her score, while Bob moves to minimize it.
Find Alice's final score.

EXPLANATION:

Let's define a *state* of the game as a pair (x, y), denoting that Alice is at vertex x and Bob is at y.
The initial state of the game is (A, B).

Let SA_u denote the sum of populations on the $A \to u$ path, and SB_u similarly denote the sum of populations on the $B \to u$ path.
These can be precomputed with DFS.

Notice that a state of the game (x, y) uniquely defines both Alice's and Bob's paths so far, and hence their scores so far.
So, for each state, it suffices for us to find the best move Alice can make.

Let $f(x, y)$ denote Alice's best score if the game starts at state (x, y).
Our objective is to compute $f(A, B)$

From a state (x, y), the next state can be any (u, v) such that:

- u is a neighbor of x and v is a neighbor of y
- u doesn't lie on the $A - x$ path and v doesn't lie on the $B - y$ path.

In particular, we can see that:

- If Alice fixes her choice of u, then Bob will choose v such that $f(u, v)$ is minimized.
- So, across all possible choices of u, to maximize her own score, Alice will choose the u such that $\min_v f(u, v)$ is maximized.

Rewriting this in terms of $f(x, y)$, we have

$$f(x, y) = \max_u(\min_v f(u, v)) + (SA_x > SB_y)$$

where the choice is across all valid neighbors u and v of x and y.

The 'brute force' method of computing this is, of course, to just iterate across all neighbors u and v of x and y and recursively compute their $f(u, v)$ values.
Let's say we also cache the values of f in a $2D$ array so that states aren't recomputed.

Let's analyze the time complexity of this.

- There are $\mathrm{O}(N^2)$ possible states. Not all of them are necessarily reachable, but the number of reachable ones can definitely be $\Theta(N^2)$.
- For each state, we do $\mathrm{O}(N^2)$ work by iterating across all pairs of neighbors, giving us a total complexity of $\mathrm{O}(N^4)$.

However, we can do a better analysis!
Notice that the transitions essentially consider a pair of edges.
However, each edge can be compared with another one at most four times, one for each pair of endpoints of the edge.
This is a tree, so there are only $(N-1)$ edges.

This means the total number of transitions we make, across all states, is bounded by $4(N-1)^2$.

In other words, our 'brute force' algorithm is really $\mathrm{O}(N^2)$, and is already fast enough!

You might notice that even caching the values of $f(x,y)$ is unnecessary, since each state is going to be visited at most once anyway.

TIME COMPLEXITY

$\mathrm{O}(N^2)$ per test case.

PROBLEM:

You're given a tree with weighted edges, and Q queries on it.
For each query:

- You're given a vertex u. Find two vertices x and y such that:
  - u lies on the path between x and y
  - The bitwise AND of the path between them is > 0
  - The distance between x and y is maximum among all such (x, y) pairs

EXPLANATION:

Let's solve for a single vertex u first.

Consider some (x, y) such that the path between them has bitwise AND > 0.
Then, there must exist some bit b such that b is set in the weight of *every* edge on this path.

So, suppose we fix a bit b and only consider those edges whose weight has b set in it.
These edges will form a forest, so consider the component (i.e tree) that contains u.
In this tree, we now just want to find the longest path that includes u.
The answer for u is then the longest such path across all b.

To restate the problem: we're given an unweighted tree and a vertex u of it. We'd like to find the longest path in this tree passing through u.

There are several ways to find this in linear time, here's a neat one.

Let Γ be the tree we're considering, and (x, y) be the endpoints of an optimal path.
Then, there exists a solution where at least one of x and y will be the endpoint of a diameter of the tree.

Proof

Let d_1 and d_2 be two endpoints of a diameter of Γ.
Recall that one property of the diameter is that the longest path from a vertex will have its other endpoint be a diameter endpoint.
In particular, the longest path from u will have either d_1 or d_2 as an endpoint. Without loss of generality, let it be d_1.

Let's root the tree at u and see what happens.

- If the path from d_1 to d_2 passes through u, clearly it's the longest path passing through u and the claim is true.
- Otherwise, there's a child c of u whose subtree contains both d_1 and d_2.
- Since (x, y) as a path passes through u, we can break the path up into disjoint $x \to u$ and $y \to u$ paths.
  - If x lies in the subtree of c, then choosing $x = d_1$ gives us a not shorter path, so we can do so.
  - If y lies in the subtree of c, similarly we can replace it with d_1 and not be worse.
  - Otherwise, x and y will lie in different subtrees of children of u. In this case, either one of them can be replaced by d_1 for a not worse answer.

We can always ensure that d_1 is one of the endpoints of an optimal path, hence proving our claim.

Now that we have this, a solution in $\mathrm{O}(N)$ is fairly simple:

- Find a diameter of Γ in $\mathrm{O}(N)$.
  There are a few different ways to do this: for example, you can use two runs of dfs/bfs as mentioned in **this article**; or you can use dp as mentioned **here** .

- Let d_1 and d_2 be the endpoints of the diameter.
- Root Γ at d_1 and compute distances to all nodes. Let this array be dist_1.
  Note that the longest path from d_1 passing through u simply equals the maximum distance of some vertex in the subtree of u, which can be computed easily using dynamic programming once dist_1 is known.
- Similarly, root Γ at d_2 to compute dist_2 and mx_2.
- The answer for u is then just $\max(mx_1[u], mx_2[u])$.

Repeat this linear algorithm for each bit to obtain a solution in $\text{O}(30 \cdot N)$, for a fixed u.

Obviously, doing the above for each query is too slow.
However, it's not really that hard to optimize this.

Notice that for a fixed bit b, the forest we end up with is independent of u.
This means that for a certain tree in this forest, if we compute the diameter and the mx_1 and mx_2 arrays mentioned above, then the answer for *every* u in this tree is $\max(mx_1[u], mx_2[u])$ — we don't need to recompute them!

This means we can fix a bit b and compute the answer for all vertices in $\text{O}(N)$ time, which in turn means we can just precompute the answer for all vertices in $\text{O}(30 \cdot N)$ time.

After this precomputation, answering queries is $\text{O}(1)$ each: just look up the answer.

There are other ways to solve this problem, for example dp with rerooting.
However, the general idea remains the same: precompute the answer for all vertices in $\text{O}(30 \cdot N)$ or similar, and use that to answer queries.

TIME COMPLEXITY

$\text{O}(30 \cdot N + Q)$ per test case

PREREQUISITES:

Euler tour of a tree, Segment trees

PROBLEM:

You're given a tree on N vertices. Vertex i has color C_i.
Let F_x denote the frequency of color x in the **entire tree**, and S_u denote the set of all colors appearing in the subtree of u.

For each vertex u, compute

$$\prod_{x \in S_1 \setminus S_u} F_x$$

modulo M.

EXPLANATION:

If M were prime, this would be a rather easy exercise in small-to-large merging.
Unfortunately, M isn't guaranteed to be prime — this throws any approaches involving division out of the window.

Let's perform an Euler tour of the tree, so that subtrees now correspond to subarrays.
Let $[L_u, R_u]$ be the subarray corresponding to the subtree of vertex u.
This makes the set of vertices *outside* the subtree correspond to the union of the ranges $[1, L_u - 1]$ and $[R_u + 1, N]$.

In particular, suppose that we are able to ensure the following:

- For any color c that appears in S_u, F_c doesn't appear outside $[L_u, R_u]$
- For any color c that doesn't appear in S_u, F_c appears exactly once outside $[L_u, R_u]$

Then we can simply take the product of the values on ranges $[1, L_u - 1]$ and $[R_u + 1, N]$ to obtain our answer!
Finding the product of a range can be done in $\mathrm{O}(\log N)$ using a segment tree, so we just need to figure out how to maintain the two invariants above.

That can be achieved with a bit of cleverness when performing a DFS.
Consider an array A of length N, such that $A_i = 1$ for all i initially.
Next, for each color x, select **exactly one** vertex u such that $C_u = x$ and set $A_{L_u} = F_x$.

This ensures that the frequency of each color appears exactly once throughout the whole array.
We only need to focus on moving it around appropriately.

Let's perform a DFS. When at vertex u with color x,

- First, recursively compute the answer for all the children of u.
- Then, suppose pos_x is the current position of F_x in the array.
  Set $A_{\mathrm{pos}_x} = 1$, and $A_{L_u} = F_x$.
  Finally, query for the product of ranges $[1, L_u - 1]$ and $[R_u + 1, N]$; the product of these two values is the answer for u.

This works because:

- First, we already ensured that each F_x value occurs exactly once in the tree.

- Next, if color x doesn't appear in the subtree of u, then F_x will surely lie outside the range $[L_u, R_u]$ and hence be included in the product.
- Finally, if color x *does* appear in the subtree of u, F_x will be set at the position of some occurrence of color x in the subtree of u — in particular, it will be inside the range $[L_u, R_u]$, and hence will not be included in the product of the outside.

Point updates and range queries can be handled in $\mathrm{O}(\log N)$ using a segment tree, and so we're done.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

Binary search, [2D prefix sums]

PROBLEM:

K students and one mentor want to stay in a hotel that has $N \times M$ rooms, arranged in N rows and M columns. The room at the intersection of the i-th row and j-th column can accommodate $A_{i,j}$ people; and the mentor can stay in the same room as a student.

The distance between two rooms (x_1, y_1) and (x_2, y_2) is $\max(|x_1 - x_2|, |y_1 - y_2|)$.
Find the minimum possible possible distance between the mentor's room and the farthest student's room.

EXPLANATION:

First, if the sum of all $A_{i,j}$ is $\leq K$, then it's impossible to accommodate $K + 1$ people in the hotel, so the answer is -1.
Otherwise, a valid answer always exists.

Suppose we fix which room the mentor is staying in, say (x, y).
Note that this is only possible when $A_{x,y} \neq 0$.
Suppose we also fix the maximum allowed distance D between the mentor and a student.

Notice that with these two constraints, the set of cells where students are allowed to stay forms a rectangular subgrid of A, specifically,

- We want all cells (i, j) such that $\max(|x - i|, |y - j|) \leq D$. This means $|x - i| \leq D$ and $|y - j| \leq D$.
- From the definition of absolute value, this means
  - $-D \leq x - i \leq D$
  - $-D \leq y - j \leq D$.
- Rearrange this to
  - $x - D \leq i \leq x + D$
  - $y - D \leq j \leq y + D$.

This gives us a range of i and j, forming the rectangle $[x - D, x + D] \times [y - D, y + D]$.
In particular, if $K + 1$ people can be fit into this rectangle, then it's possible for the maximum distance to be *at most* D.

Checking whether $K + 1$ people fit into this rectangle is simple to do in $\mathrm{O}(1)$ after some precomputation.
Notice that we only want the sum of all values in the specified rectangle. This is doable with 2D prefix sums: a tutorial can be found [here].

We are now able to quickly check, for a fixed (x, y) and D, whether a maximum distance of $\leq D$ is possible.
However, there are $N \times M$ possible cells (x, y) and upto $\max(N, M)$ values of D for each of them, so going through them all would still be too slow.

However, notice that if we're able to achieve a maximum distance of $\leq D$, then of course we can achieve a maximum distance of $\leq D + 1$.
So, we only need to find the *smallest* D such that there exists *some* cell (x, y) which satisfies the condition.

This is exactly what binary search does!

That gives us the following solution:

- Binary search on the value of D, from 0 to $\max(N, M)$.

- For a fixed value of D, go through all cells (x, y) such that $A_{x,y} \neq 0$, and check whether any of them allow for a maximum distance of $\leq D$, using 2D prefix sums as discussed above.

For a fixed value of D, this takes $\mathrm{O}(NM)$ time.
Since we're applying binary search, we check only $\mathrm{O}(\log \max(N, M))$ values of D, for a solution that's $\mathrm{O}(NM \log \max(N, M))$.

TIME COMPLEXITY

$\mathrm{O}(NM \log \max(N, M))$ per test case.

PROBLEM:

You're given two arrays A and B, each containing integers between 1 and K.
In one move, you can insert an element between 1 and K into either A or B.
FInd the minimum number of moves such that the average of A is **strictly larger** than the average of B.

EXPLANATION:

First, if $K = 1$ then A and B can both contain only 1's, and hence will have an average of 1 no matter what.
So, making A have a strictly larger average is impossible, and the answer is -1.

If $K > 1$, it's always possible to do so.

As a preliminary observation, note that it's never optimal to insert anything less than K into A, and anything more than 1 into B.

So, suppose we insert x elements into A and y elements into B.
If this makes the average of A larger than the average of B, we'll have

$$\frac{A_1 + A_2 + \ldots + A_N + x \cdot K}{N + x} > \frac{B_1 + B_2 + \ldots + B_M + y}{M + y}$$

Our aim is to minimize $x + y$, while ensuring that the above inequality holds.

Suppose we fix the value of x.
Then, the left side of that inequality becomes a constant, say C_x.
We'd then like to find the smallest possible value of y such that $C_x > \frac{B_1 + \ldots + B_M + y}{M + y}$

This can be found by direct math by manipulating the above inequality, or by using binary search (because the average of A is now fixed, and increasing y means adding more 1's to B, which can only decrease its average).
Either way, for a fixed x the problem is solved in $\mathrm{O}(1)$ or $\mathrm{O}(\log(\text{something}))$.

In fact, we don't really have to check for too many values of x.
Note that if we choose $x = M + 1$ and $y = N + 1$, we have:

- The average of A is $\dfrac{A_1 + \ldots + A_N + K \cdot (M + 1)}{N + M + 1}$
- The average of B is $\dfrac{B_1 + \ldots + B_M + N + 1}{N + M + 1}$
- Denominators are equal, so it's enough to compare their numerators.
- We have:
  - $A_i \geq 1$ for each i, so $A_1 + \ldots + A_N \geq N$
  - $B_i \leq K$ for each i, so $B_1 + \ldots + B_M \leq K \cdot M$
  - $K > 1$
- Putting all three together, we have $A_1 + \ldots + A_N + K \cdot (M + 1) > B_1 + \ldots + B_M + N + 1$, which is exactly what we wanted.

So, $x = M + 1$ and $y = N + 1$ gives us a solution already.
In particular, this means that $N + M + 2$ is an upper bound for $x + y$.

This means it suffices to check for each x from 0 to $N + M + 2$ what the best y is, and take the minimum across them all.
Note that this also means the binary search for y can be done in $\mathrm{O}(\log(N + M))$.

Since each x is processed quickly, this solution is fast enough.

TIME COMPLEXITY

$\mathrm{O}(N + M)$ or $\mathrm{O}((N + M)\log(N + M))$ per test case.

PREREQUISITES:

Dynamic programming

PROBLEM:

You have two counters a and b and a score S, all initially 0.
For N minutes, you need to increase a or b by 1.
There are also M events, each of one of two types:

- A type 1 event increases S by 1 if $a > b$, and sets $a = b = 0$ otherwise.
- A type 2 event increases S by 1 if $a < b$, and sets $a = b = 0$ otherwise.

What's the maximum possible final score you can achieve?

EXPLANATION:

Let $d = a - b$ denote the difference between a and b.
Note that each event checks whether the difference is positive or negative, then either increases the score by 1 or resets d to 0.

Also, in each minute, we can either increase or decrease d by 1.

Now, ideally we'd like to keep d somewhat close to 0 so we can quickly flip its sign if needed for an event.
In fact, it's enough to keep $-2 \le d \le 2$ at all times.

Proof

Consider some optimal sequence of moves that has $|d| > 2$ at some point. Without loss of generality, let's say d reaches 3. We can construct a new sequence of operations as follows:

Consider the first time we make the move $2 \to 3$.
Instead, let's make the move $2 \to 1$ in this step. Note that this essentially subtracts 2 from all later values of d.

Now,

- If the original sequence of moves never has the move $3 \to 2$ after this point, then we don't modify anything more.
  - The original sequence and new sequence both remain always positive after the change, and are the same before the change, so the new sequence has the same score as the original.
- If the original sequence of moves has the move $3 \to 2$ at some point, it corresponds to the move $1 \to 0$ in the new sequence.
  - Replace the first instance of such a move with the move $1 \to 2$ in the new sequence. Once again, it's not hard to see that the score doesn't change.

By continually repeating this process, we can ensure the entire sequence has $|d| \le 2$ at all times while still remaining optimal.

Armed with this knowledge, let's actually solve the problem.

We'll use dynamic programming: let $dp(i, x)$ denote the maximum score after i minutes if $d = x$.
Transitions are as follows:

- First, we need to either increase or decrease d by 1. That is, $dp(i, x) = \max(dp(i-1, x-1), dp(i-1, x+1))$.
- Then, we need to process an event at this time point, if there is one.
- Suppose there is a type 1 event. That changes the values as follows:

  - If $x > 0$, then add 1 to $dp(i, x)$
  - $dp(i, 0) = \max(dp(i, 0), dp(i, -1), dp(i, -2))$ since negative values are forced to 0
  - $dp(i, x) = -\infty$ for $x < 0$, since it's impossible to have a negative value of d after this minute.
- A type 2 event can be handled similarly.

The base cases for this function are $dp(0, 0) = 0$ and $dp(0, x) = -\infty$ for $x \neq 0$.
The final answer is the maximum value of $dp(N, x)$ for some x.

Since only $-2 \leq x \leq 2$ matters, we have $5N$ states and $\mathrm{O}(1)$ transitions for each, leading to a $\mathrm{O}(N)$ solution.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

Details

Let's perform our DFS, and suppose we're at vertex u. We want to find $\text{val}[u]$, the answer for u within its subtree.
Let c be a child of u, and w be the weight of the edge joining them. First, recursively solve for c to compute $\text{val}[c]$.

Now, let's consider all paths from u going into the subtree of c.
Each such path corresponds to a path from c going into its subtree.
Specifically, we can take any path starting at c and going down, circular rotate every edge weight on this path once (since they'll all be one step further from u than c), and then xor with w (the weight of the $u \to c$ edge).

However, it's not immediately obvious how to quickly do this for even a single path, let alone all paths.

Instead, let's look at what happens bit-by-bit.
Let $\text{ct}[u][b]$ be the number of paths from u into its subtree that have bit b set.
If we are able to calculate $\text{ct}[u][b]$ for each $0 \le b < 32$, then we'll simply have $\text{val}[u] = \sum_{b=0}^{31} \text{ct}[u][b] \cdot 2^b$.

Let's recursively compute $\text{ct}[c][b]$ for the child c of u.
Then, notice that bit b in a $c \to v$ path corresponds to bit $(b+1)$ (modulo 32) in the $u \to v$ path, since the distance will increase by one.
In particular,

- If w has bit b *unset*, then $\text{ct}[u][b]$ increases by $\text{ct}[c][b-1]$.
- if w has bit b *set*, then $\text{ct}[u][b]$ increases by $\text{sz}[c] - \text{ct}[c][b-1]$, where $\text{sz}[c]$ denotes the subtree size of c.
  This is because we're XOR-ing with w, so any path with the bit set till c will have it unset when moving to u, and vice versa.

In this way, we can compute the contribution of child c to vertex u in $\text{O}(32)$.

This results in an algorithm in $\text{O}(32 \cdot N)$ in total.

Now that we've computed the subtree answer for each u, we need to reroot.

The rerooting process follows basically the same procedure.
When moving from u to its child c, we have the following:

- The rerooting procedure has already computed the path counts for each bit for paths starting at u that are outside it. We want to extend this to path counts for c.
- First, we need to account for all the children of u that are not c, since when moving to c, they'll be outside its subtree.
  This is not hard to do: after all, our initial DFS already explicitly computed the path counts from u to each of these other children, so all we need to do is take those values and remove the ones corresponding to child c, i.e, reverse the process for child c.
- Now, we know the path counts for everything from u. We want to shift this to path counts to everything outside c via edge weight w, which is exactly the same algorithm as we used when moving from child to parent, and hence is done in $\text{O}(32)$.

We DFS twice, each with a complexity of $\text{O}(32N)$, which is fast enough for the given constraints.

TIME COMPLEXITY

$\text{O}(32 \cdot N)$ per test case.

PROBLEM:

You are given an array A, each of whose elements has frequency ≤ 100.

Count the number of subarrays whose mode is unique.

EXPLANATION:

Each element has frequency ≤ 100, which means the mode of any subarray also has a frequency of at most 100.

Let's fix the right endpoint R of our subarray, and the frequency of the mode k. Let's try to compute the number of L such that subarray $[L, R]$ has a unique mode with frequency k.

The major observation to be made is that the set of valid L form a contiguous sequence, i.e, the set of valid L is $\{x, x+1, \ldots, y-1, y\}$ for some x and y; so let's instead try to find x and y.

Proof

Suppose we start L at R and keep decreasing it. As we move from subarray $[L, R]$ to $[L-1, R]$, the frequency of the mode can never decrease; and for a fixed frequency, the number of modes also can never decrease.

So, y is the first time the frequency of the mode is k, and x is the last time the frequency is k and there is only a single mode.

Finding x and y in $\mathrm{O}(N)$ each is not too hard (but of course, too slow). When iterating in *reverse* down from R:

- y is the first index at which some element occurs exactly K times.
- $x-1$ is the higher of:
  - The first index at which some element occurs exactly $K+1$ times (increasing the frequency of the mode); or
  - The second index at which some element occurs exactly K times (increasing the number of modes)

To improve this, look at what information exactly we need.
We need to know the first time something occurs K times, the first time something occurs $K+1$ times, and the second time something occurs K times. We need this information for every $1 \leq K \leq 100$.

So, let's try to maintain this information. The array is treated as 1-indexed here.

- Let $\text{first}[K]$ denote the first time $\leq R$ that an element occurs K times.
- Let $\text{second}[K]$ denote the second time $\leq R$ that an element occurs K times.
- For convenience, let's say $\text{first}[K]$ and $\text{second}[K]$ are 0 if no valid position exists.

When we move from R to $R+1$, this information changes as follows:

- For elements other than A_{R+1}, there is of course no change in their closest positions; so it suffices to look at the last 100 occurrences of A_{R+1} alone.
- Consider the K-th previous occurrence of A_{R+1}. Suppose this is position u.
  - If $u > \text{first}[K]$, then set $\text{second}[K] \leftarrow \text{first}[K]$ and $\text{first}[K] \leftarrow u$
  - Otherwise, set $\text{second}[K] \leftarrow \max(\text{second}[K], u)$
- This allows us to update the first and second arrays in 100 operations.
- Now, for a fixed K, we have $y = \text{first}[K]$ and $x = 1 + \max(\text{first}[K+1], \text{second}[K])$, after which we simply add $y - x + 1$ to the answer.

The above algorithm only needs us to find the previous 100 occurrences of A_{R+1} quickly, which is easy if we create a list of positions corresponding to each value.

This gives us a solution in $\mathrm{O}(N \log N + 100 \cdot N)$ or $\mathrm{O}(100 \cdot N \log N)$, depending on implementation.

TIME COMPLEXITY:

$\mathrm{O}(N \cdot 100)$ per testcase.

PROBLEM:

You're given a directed graph with N vertices and M edges.
The *cost* of a subset of vertices is defined to be the number of pairs of vertices in the subset that aren't mutually reachable from each other.

Answer Q queries.
For each query, given K, find the maximum size of a subset whose cost doesn't exceed K.

EXPLANATION:

The cost of a subset S is the number of pairs of vertices in the subset that aren't mutually reachable from each other.
Turning that around, if S has size M and there are x pairs of vertices in S that *are* mutually reachable from each other, then $cost(S) = \binom{M}{2} - x$.

The latter condition is a bit simpler to deal with because it's more familiar.
In particular, we know that in a directed graph, two vertices are mutually reachable from each other *if and only if* they lie in the same **strongly connected component** (henceforth, SCC).
This hints to us that we should look at the SCCs of the given graph.

In particular, notice that we can take *all* the vertices from any single SCC, for zero cost.
However, if we have vertices from different SCCs, the cost will always be positive—intuitively, it seems better to not choose vertices from too many different SCCs, because the cost is just the number of pairs of vertices belonging to different SCCs.

Notice this in fact tells us the answer for $K = 0$: the size of the largest SCC.
Once again, this is a hint that the sizes of the SCCs are also important, not just the SCCs themselves.

So, let's first find all the SCCs of the given graph, and also their sizes.
This can be done in $\mathrm{O}(N + M)$ time using some different algorithms: for example, **Kosaraju's algorithm** or **Tarjan's algorithm**.

Suppose there are r SCCs, denoted $C_1, C_2, \ldots, C_r$.
Let their sizes be $s_1, s_2, \ldots, s_r$; and for convenience, let's have $s_i \geq s_{i+1}$.

Our look at $K = 0$ above gives us a natural-sounding greedy algorithm:

- First, take all the vertices belonging to C_1, the largest SCC, for a cost of 0.
- Then, take as many vertices as we can from C_2, as long as the cost doesn't exceed K.
- Then, do the same to $C_3, C_4, \ldots$ in order.

That is, take vertices from SCCs in descending order of size.

It turns out that this greedy is indeed optimal!

Proof

Since there are multiple queries, we need to be able to perform this greedy algorithm quickly.
To do that, notice that our solution has a somewhat predictable structure:

- First, we take all vertices from SCCs $C_1, C_2, \ldots, C_l$ for some l.
- Then, we take some (but not all) vertices from C_{l+1}

Let's compute for each part separately.

First, we need to find the largest l so that the cost of taking the SCCs $C_1, C_2, \ldots, C_l$ doesn't exceed k.
The cost of this choice is the number of pairs of vertices belonging to different SCCs.

Since we know sizes already, this is easily seen to be $\sum_{i=1}^{l} \sum_{j=i+1}^{l} s_i s_j$.
This cost can in fact be precomputed using prefix sums, as follows:

- Let P_i denote the cost of taking the first i SCCs.
- Then, $P_i = P_{i-1} + s_i \cdot (s_1 + s_2 + \ldots + s_{i-1})$.
- So, by maintaining prefix sums of the P and s arrays, all costs can be computed in $\mathrm{O}(r)$ time.

Once costs are precomputed, finding the largest l is easy: just binary search on these costs to find the largest l such that $P_l \leq K$.

Next, we can still take some vertices from C_{l+1} (if it exists).
We've already incurred a cost of P_l so far, so we're left with $K - P_l$ to work with.
Note that each new vertex we take increases the cost by exactly $s_1 + s_2 + \ldots + s_l$.
So, the number of vertices we can take is exactly $\lfloor \frac{K-P_l}{s_1+s_2+\ldots+s_l} \rfloor$.

That's the entire solution!
After some precomputation, each query is answered in $\mathrm{O}(\log N)$ time since we only perform a single binary search.

TIME COMPLEXITY

$\mathrm{O}(N \log N + M + Q \log N)$ per testcase.

PREREQUISITES:

Greedy/dynamic programming, careful implementation

PROBLEM:

You're given an array A. Find the minimum number of elements that need to be deleted from it so that the remaining array satisfies

$$A_1 + A_2 = A_3 + A_4 = A_5 + A_6 = \ldots = A_{2k-1} + A_{2k}$$

EXPLANATION:

The final array must be such that it can be partitioned into k adjacent pairs, each having the same sum.
So, let's fix what the final sum is, then try to find the longest array we can form that satisfies the condition.

Suppose we've fixed the sum to be S.
Let pairs_S be a list consisting of all pairs (i,j) such that $A_i + A_j = S$.
Note that pairs_S can be precomputed and stored for *all* S by simply iterating over all pairs of elements in A.

Suppose we choose to keep the pair (i,j) in the final array.
Then, notice that any other pair (i_2,j_2) must satisfy $i_2 > j$ **or** $j_2 < i$.
That is, we must have either $i_2 < j_2 < i < j$ or $i < j < i_2 < j_2$, since we're pairing i and j and hence can't have i_2 or j_2 lie between them.

Thinking of it another way, we can consider the pair (i,j) to be the *interval* $[i,j]$. The above condition then means that any intervals we pick must be disjoint.

Our problem thus transforms to: given a list of intervals, find the maximum number of disjoint intervals that can be picked from them.

This is a classical problem, and can be solved greedily as follows:

- Sort the intervals in increasing order of their **right endpoints**.
  That is, $[l_1, r_1]$ comes before $[l_2, r_2]$ if $r_1 < r_2$ (doesn't really matter how ties are broken).
- Let $R_{\max}$ denote the largest right endpoint we've encountered so far.
  Initially, $R_{\max} = -\infty$
- Then, for each interval $[l, r]$ in this sorted list:
  - If $l \le R_{\max}$, ignore it
  - Otherwise, $[l, r]$ is disjoint from all the intervals seen so far, so take it into the answer and set $R_{\max} := r$

This greedy strategy is **provably optimal**.

If pairs_S has K intervals in it, this algorithm takes $\mathrm{O}(K \log K)$ time, but that's only because we need to sort the intervals since the greedy part is linear.
In the specific case of this problem, we can in fact create the intervals in sorted form, eliminating the need for sorting entirely.

That can be done by the following pseudocode:

```
for j in 1...n:
    for i in 1...j-1:
        pairs[a[i] + a[j]].append((i, j))
```

That is, by iterating over the right endpoint in increasing order and then iterating over the left endpoint.

The time complexity of this approach is $\mathrm{O}\left(\sum_S len(\mathrm{pairs}_S)\right)$, which is just $\mathrm{O}(N^2)$ since each pair (i,j) appears in exactly one sum.

Since the values of $A_i + A_j$ can be as large as $2 \cdot 10^9$, pairs will likely need to be a map of some kind. Depending on the kind of map used, this adds an $\mathrm{O}(\log N)$ or (expected) $\mathrm{O}(1)$ to the time complexity. Hashmaps (`unordered_map` or `gp_hash_table` in C++, `dict` in Python) in particular pass well within the time limit.

There are other solutions to this problem, for example using dynamic programming.
However, you'll need to be reasonably careful when implementing such solutions, since too many map accesses or extra log factors will result in TLE.

TIME COMPLEXITY

$\mathrm{O}(N^2 \log N)$ or expected $\mathrm{O}(N^2)$ per test case

PROBLEM:

Given a string S of length N, find the number of ways of partitioning it into substrings such that each resulting substring has exactly k vowels.

EXPLANATION:

Let pos_i denote the position of the i-th vowel in string S.

Let's look at how we can partition S into substrings containing exactly k vowels.

- The first k vowels must be in the same substring; and this substring cannot include the $(k+1)$-th vowel.
  So, the first 'cut' must be made after pos_k, but before pos_{k+1}.
  There are $\text{pos}_{k+1} - \text{pos}_k$ such choices.
- The next consecutive k vowels must be in the same substring.
  By the same reasoning as above, the second cut must thus be made after pos_{2k}, but before pos_{2k+1}.
  There are $\text{pos}_{2k+1} - \text{pos}_{2k}$ choices here.
- More generally, it's easy to see that for the i-th cut, there are $\text{pos}_{ik+1} - \text{pos}_{ik}$ choices.

Finally, note that each cut is independent of the others.
So, the total number of ways is just the product of all these numbers.

In particular, if the string has $x \cdot k$ vowels, we need exactly $x - 1$ cuts to separate them.
So, the answer will be

$$(\text{pos}_{k+1} - \text{pos}_k) \cdot (\text{pos}_{2k+1} - \text{pos}_{2k}) \cdot \ldots \cdot (\text{pos}_{(x-1)k+1} - \text{pos}_{(x-1)k})$$

The pos_i values can be computed by just iterating across the string, so this algorithm takes $\text{O}(N)$ time.

TIME COMPLEXITY

$\text{O}(N)$ per testcase.

PROBLEM:

Given L and R, find the number of integers x such that there exist $L \le a \le b \le R$ satisfying $a + b = a \oplus b = x$.

EXPLANATION:

First off, $a + b = a \oplus b$ means that a and b must have completely disjoint binary representations, i.e, $a \ \& \ b = 0$.
With that in mind, let's go over the subtasks.

Subtask 1

The limits here are very small, simply iterate over all pairs of (a, b), and if they satisfy the condition insert their sum into a set.
The answer is the size of the set.

The trivial implementation of this takes $\mathrm{O}((R - L)^2 \log (R - L))$ time, which is good enough.

Subtask 2

Here, we have $L = 0$ and $R = 2^k - 1$, i.e, all numbers with at most k significant bits.
It's possible to make every number from L to R here, since to make x we can choose $a = 0$ and $b = x$.

Making anything $> R$ is impossible, because the xor of two k-bit numbers cannot have $> k$ significant bits.

So, the answer is the length of the range $[0, R]$, which is $R + 1$.
Note that $R + 1 = 2^k$, so the answer essentially says we can make all k-bit strings.

Subtask 3

Here, we still have $L = 0$, though we might not have $R = 2^k - 1$.

With the same logic as the previous subtask, we can certainly make all x in the range $[0, R]$.
However, we might now be able to reach $x > R$ as well (for example, if $R = 2$ we can reach $3 = 1 + 2 = 1 \oplus 2$).

In fact, if R is a k-bit number, we can reach all the k-bit numbers!

Proof

Let R be a k-bit number, i.e, k is the smallest integer such that $R \le 2^k - 1$.
Then, certainly we must have $2^{k-1} \le R$.

Any k-bit integer $x \ge 2^{k-1}$ can be reached by choosing $b = 2^{k-1}$ and $a = x - b$.
Note that this works because $a < 2^{k-1}$.

Numbers $\le R$ can be formed trivially by $(0, x)$ as noted eariler; and this method allows us to form all the numbers from $R + 1$ to $2^k - 1$, so we're done.

This gives us a simple solution: if R is a k-bit integer, the answer is just 2^k.

Finding k requires us to find the highest set bit in R, which can be done by just iterating k from 0 — it'll take $\mathrm{O}(\log R)$ time which is good enough.

Subtask 4

The constraints here allow for a linear (or close to linear) solution.

Note that any valid x must be $\leq 2R$.
So, let's iterate over each x from 0 to $2R$ and see if we can make it.
If $x = 0$, then we can make it with $(0, 0)$ if $L = 0$, and can't make it otherwise.

If $x > 0$, then we can write its binary representation as $x = 2^{p_1} + 2^{p_2} + \ldots + 2^{p_m}$ where $p_1 > p_2 > \ldots > p_m$.
To write $x = a \oplus b = a + b$, we need to distribute the bits $p_1, p_2, \ldots, p_m$ to a and b.
Clearly, since b is the larger one, it must receive 2^{p_1}.
After this, it's best to give all remaining 2^{p_i} to a, since:

- Increasing b too much might make it exceed R
- Keeping a too small might leave it $< L$

So, the best chance we have is with $a = 2^{p_2} + 2^{p_3} + \ldots + 2^{p_m}$ and $b = 2^{p_1}$.
That is, $b = 2^{p_1}$ and $a = x \oplus b$.
Form these a and b, and check if they both lie in $[L, R]$.
If they do, increase the answer by 1.

The check for each x can be done in $\mathrm{O}(1)$ or $\mathrm{O}(\log x)$, and we check upto $2R$ different values of x so this takes $\mathrm{O}(R \log R)$ time, which is good enough.
Subtask 5

Now, L and R are too large for any linear-type solutions to work.

Let's look back at how we solved subtask 4:

- We fixed a value of x between 1 and $2R$ (0 was special-cased out)
- We found the highest bit of x and gave it to b; and gave all the other bits of x to a
- Then, we checked if this a and b were in range $[L, R]$.

Notice the second point in particular: the only b we care about are powers of 2.
So, let's fix b as a power of 2 in the range $[L, R]$ and see which values of a are valid, as per our algorithm.

- First off, by our construction we must have $a < b$.
- We also must have $a \geq L$.
- It's not hard to see that *any* such a is valid; i.e, we can choose any a in the range $[L, b - 1]$.
  The length of this range is $b - L$.

This gives us a pretty simple algorithm:

- Fix the value of b to some power of 2 that lies in $[L, R]$.
- Then, add $b - L$ to the answer.

If $L = 0$, add 1 to the answer since we can also form $x = 0$.

Since we only care about powers of 2 that don't exceed R, we check $\log R$ of them.
So, this algorithm takes $\mathrm{O}(\log R)$ time, which is fast enough to get AC.

TIME COMPLEXITY

$\mathrm{O}(\log R)$ per test case.

PREREQUISITES:

Bitwise operations

PROBLEM:

Given distinct integers A, B, C, find any $0 \le X < 2^{30}$ such that

$$A \oplus X < B \oplus X < C \oplus X$$

or claim that none exist.

EXPLANATION:

Let's just look at A and B first.

Looking at their binary representations, let k be the *highest* bit where they differ.
For example, if $A = 28 = 11100_2$ and $B = 26 = 11010_2$, we'd have $k = 2$, because the first time the binary representations differ is at $2^2 = 4$.

Let a_k be the value of this bit in A, and b_k be the value of this bit in B.
The direction of inequality between A and B is completely defined by what a_k and b_k are; in particular, we want $a_k = 0$ and $b_k = 1$ to hold.

So,

- if $a_k = 0$ and $b_k = 1$, then the k-th bit of X *must* be 0
- if $a_k = 1$ and $b_k = 0$, then the k-th bit of X *must* be 1

This fixes one of the bits of X.

Do the same for the pairs (B, C) and (A, C) as well, giving us three conditions on the bits of X.

If any of these conditions conflict (for example, if (A, B) says that the 10-th bit must be 0 while (B, C) says that it must be 1), no answer is possible; so print -1.

Otherwise, fix these three bits of X, and the others can be left as zeros since there are no constraints on them; so this gives us a valid value of X.

The fact that at most three bits really matter also gives us a rather funny randomized solution: generate a random X between 0 and $2^{30} - 1$ and check if it satisfies the condition; and if it doesn't, continue generating X till you find a valid one.
If no valid X is found after several attempts (say, 100 attempts), the answer is -1.
Do you see why this works with high probability?

TIME COMPLEXITY

$\mathrm{O}(1)$ per test case.

PREREQUISITES:

Familiarity with bitwise XOR

PROBLEM:

There's a hidden array A of length N.

You can provide the judge with two values i and X, following which the judge will set A_i to $A_i \oplus X$.
Then, the judge will tell you the maximum element of the array.

Use at most 10^5 operations to make all the elements 0.

EXPLANATION:

The constraint on A_i being 2^{30} and the fact that $30 \cdot N \leq 10^5$ (since $N \leq 3000$), along with the operation being a bitwise one, should hint at doing something bit-by-bit.

I'll denote the operation where A_i is xor-ed with X as $\text{query}(i, X)$.

We have no information to start with, other than the fact that the elements are distinct.
Let's analyze what happens we perform a query.

- First, we can obtain the initial maximum of the array by choosing $X = 0$, since this doesn't change any element.
- Now, let's see what happens to the maximum when we do $\text{query}(i, X)$.
  - If the maximum doesn't change, we don't get too much information.
  - If the maximum does change, we in fact get a lot of information!
    - If the maximum reduces, then A_i used to be the maximum. This means the current value of A_i is $\text{prev_max} \oplus X$, so performing $\text{query}(i, \text{prev_max} \oplus X)$ sets A_i to zero.
    - If the maximum increases, the A_i is the new maximum, so performing $\text{query}(i, \text{new_max})$ sets A_i to zero.

Either way, if we perform a query and the maximum changes, we can set the current position to zero: which means we never need to both with it in the future.
This should give us some idea: we just need to be able to trigger this change at least once for each index, and we'll be done.

Combined with the 'look at bits' intuition from the start, here's one way to do that:

- Let's iterate over b from 0 to 29. Fix some i such that $A_i \neq 0$.
- Do $\text{query}(i, 2^b)$. If this changes the maximum, set A_i to 0 as discussed above; otherwise do nothing.
- There's one caveat: if $b = 0$, run this loop from 1 to N *twice*.
  - This is only really needed for the case when the initial maximum is $2^{30} - 1$, so other ways of dealing with that case work too: for example, choose a random integer x and operate it on every array element (which works with reasonably high probability)

This obviously uses a small enough number of operations: we loop across the entire array 31 times, and then use at most one more operation for each index to set it to zero, for a total of $\leq 32N$ operations, which is within 10^5 for $N \leq 3000$.

We only need to prove that it is indeed correct, i.e, sets everything to zero.

Proof

Let M denote the maximum value of the array. Note that M does change as we perform operations.
We do a bit of casework on the value of M.

First, if $M < 2^{29}$, then our algorithm is easily seen to be correct: no matter how the lower bits change during our operations, $A_i \bigoplus 2^{29} = A_i + 2^{29}$ will always be the maximum element, so the last iteration of the loop will set everything to zero.
Now, we deal with $M \geq 2^{29}$.

Next, let's consider the case when $M < 2^{30} - 1$, i.e, M doesn't have all its bits set.
In particular, let k be the *highest unset bit* in M.
Once we've performed operations on bits $0, 1, 2, \ldots, k-1$, the maximum may have changed; but it's still true that the current maximum cannot have all the bits $29, 28, \ldots, k+1, k$ set.

Now, let's perform $\text{query}(i, 2^k)$ for some i. Of course, if we can set A_i to zero after this, we do so.
Notice that (before the operation), if A_i has bits $29, 28, \ldots, k+1$ set then k must be unset.
$\text{query}(i, 2^k)$ then sets this bit, making A_i the unique largest element of the array (and hence we can immediately set it to zero).

This means the only remaining non-zero elements are those with at least one bit from $29, 28, \ldots, k+1$ missing; and so we've increased the largest missing bit by at least 1.
Since we're iterating from lower to higher, these elements will also eventually be set to zero.

Finally, we're left with the case when $M = 2^{30} - 1$, when we can't choose the largest unset bit to operate on.
If we can somehow make $M < 2^{30} - 1$, we'll be happy, since we have an algorithm for that case.
Here's where we use the fact that all the elements are distinct.

Suppose $A_m = 2^{30} - 1$.
Notice that performing any non-zero move on position m will reduce it, hence changing the maximum.
So, if we do $\text{query}(i, 1)$ for every i, we'll eventually change the maximum.

However, there's one small pitfall here: if there exists a $j < m$ such that $A_j = 2^{30} - 2$, then $\text{query}(j, 1)$ will set $A_j = 2^{30} - 1$; and then when we change A_m the maximum doesn't actually change, so we're back where we're started.

A simple way to solve this is to simply run $\text{query}(i, 1)$ on every index again!
This works because if the maximums did swap position as mentioned above, it can't do that a second time (since $2^{30} - 2$ now occurs after $2^{30} - 1$).
Also, if the maximum after the first loop is no longer $2^{30} - 1$, then the second loop cannot create $2^{30} - 1$ as the maximum — it can be seen that if the loop removes $2^{30} - 1$ as the maximum, then $2^{30} - 2$ cannot be the new maximum at the end of it.

Alternately, you can also just pick a random integer X and do $\text{query}(i, X)$ for every i.
With probability at least $1 - \frac{N}{2^{30}} \geq 0.999997206$, you pick an X such that $2^{30} - 1 - X$ isn't in the array, which means no collisions can occur.

TIME COMPLEXITY:

$\text{O}(N \log 2^{30})$ per testcase.

PROBLEM:

You have an array A. In one move, you can pick an index i (with $1 \le i < N$), an integer X, and set $A_i \to A_i \oplus X$ and $A_{i+1} \to A_{i+1} \oplus X$.

Determine whether all the elements of A can be made equal.

EXPLANATION:

The main observation behind solving this task is the fact that the bitwise XOR of the entire array doesn't change after an operation is performed.

Proof

Let $S = A_1 \oplus A_2 \oplus \ldots \oplus A_N$.
Suppose we perform an operation on positions $(i, i+1)$ with value X. The bitwise XOR of the new array is $A_1 \oplus A_2 \oplus \ldots \oplus (A_i \oplus X) \oplus (A_{i+1} \oplus X) \oplus \ldots \oplus A_N$.

This equals $S \oplus X \oplus X = S$.

Now, suppose we were able to make our final array such that $A_i = y$ for every $1 \le i \le N$ (where y is some integer).
The bitwise XOR of this array is $\underbrace{y \oplus y \oplus y \oplus \ldots \oplus y}_{N \text{ times}}$.

- If N is even, this is simply 0 because $y \oplus y = 0$.
  - So, if N is even, the bitwise XOR of the original array *must* be 0 for the answer to be YES.
- If N is odd, this is y.
  - This gives us no further restrictions, since we can choose y to be the bitwise XOR of the whole array and the equation is satisfied.

In fact, this solves the problem!

- If N is odd, the answer is always YES.
- If N is even, the answer is YES if and only if the bitwise XOR of the whole array is 0.

Proof

Consider the case when N is odd. Let y be the bitwise XOR of the whole array.
Let's make A_i equal to y for each i from 1 to N in order.

For each i from 1 to $N-1$:

- Let $X = A_i \oplus y$. Perform the operation on $(i, i+1)$ with this X.
  - Note that A_i changes to $A_i \oplus X = A_i \oplus A_i \oplus y = y$.
- This allows us to make all of $A_1, A_2, \ldots, A_{N-1}$ equal to y.
- This automatically makes A_N equal to y.
  - This follows from the fact that the bitwise XOR of the whole array must be y; and the bitwise XOR of the first $N-1$ elements is 0 (since it's an even number of copies of y).
  - So, the remaining element must be y.

The same construction works for the case when N is even and the bitwise XOR of the whole array is 0: just make the first $N-1$ elements 0, and the last element automatically becomes 0.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.

PREREQUISITES:

Sorting

PROBLEM:

Given an array A, in one move you can choose $x, y \in A$, delete them, and add $x \oplus y$ to A.
Maximize the product of A.

EXPLANATION:

A little analysis of how the operation affects the product of all the elements should tell you that it's almost never optimal to replace two elements.

In fact, it's optimal to replace x and y with $x \oplus y$ *if and only if* $x = 1$ and y is even.

Why?

This can be seen somewhat intuitively by looking at just two elements.
Let $x \leq y$, and we want to decide whether to operate on x and y or not.

First, note that we have $x \oplus y < 2y$, which should both be obvious to see by looking at the bits independently.
Now,

- If we don't operate on x and y, we contribute xy to the product.
- If we do operate on them, we contribute $x \oplus y < 2y$ to the product.

In particular, if $xy \geq 2y$, i.e, $x \geq 2$, then it's always better to *not* perform the operation.
This forces $x = 1$.

Now we have to decide which y give us $1 \oplus y > 1 \cdot y = y$. Note that:

- If y is even, $1 \oplus y = y + 1$
- If y is odd, $1 \oplus y = y - 1$

This tells us that $x = 1$ and y being even is the *only* optimal case to perform an operation on two elements.

It's somewhat reasonable to expect this to hold when we need to perform more than one move, but a lot less obvious why: after all, the order of moves matters, and maybe we want to perform one 'bad' move to be able to get to a 'good' one later.
It so happens that this is never the case. A slightly more detailed proof is attached below if you're interested.

More detailed proof

Let $B = [B_1, B_2, \ldots, B_k]$ be the final array, after we have performed some operations.
Note that $B_i = A_{i_1} \oplus A_{i_2} \oplus \ldots A_{i_r}$ for some indices $i_1, \ldots, i_r$.
Let's call each A_{i_j} a *component* of B_i.

Suppose there exists a B_i such that it has at least two components that are ≥ 2. W.l.o.g let $A_{i_1} \geq 2$.
Then, we can instead perform operations so that we end up with $B_i \oplus A_{i_1}$ and A_{i_1} instead of just B_i; and this gives us a strictly higher product.
So, an optimal solution will never have such a B_i.

Now suppose some B_i has ≥ 2 components that are 1.
Then, we can remove two ones from this component (which doesn't change its xor) and keep those ones as two more separate components: this doesn't affect the product.
So, there exists an optimal solution in which each B_i has at most one 1, and at most one value ≥ 2.

Now consider $B_i = 1 \bigoplus y$ where $y \geq 2$.
As we noted above, if y is odd it's better to have y and 1 separately.

So, an optimal solution can only have B_i that are either single elements, or $1 \bigoplus y$ for even y.
This completes the proof.

With this information in hand, let's now get to actually solving the problem.

We can simply simulate the process: as long as we have at least one 1 and one even number remaining, perform an operation on them.
All that remains is to decide which even number to operate on at each step. This is simple: choose the smallest remaining even number.

Why?

Note that we're choosing $x = 1$ and even y, which means we're removing $1 \cdot y$ from the product and multiplying it by $y + 1$ instead. So, our product is multiplied by $\frac{y+1}{y}$.

$\frac{y+1}{y}$ is larger the smaller y is, so it's optimal to choose the smallest y we can (while ensuring it's even).

Implementing this is fairly straightforward: count the number of 1's in the sequence, then sort the even numbers and keep choosing the smallest one of them to operate on while there are remaining 1's.
Note that operating on an even number turns it into an odd number so the list doesn't need to be updated.

TIME COMPLEXITY:

$\mathrm{O}(N \log N)$ per testcase.

PROBLEM:

You have an array A of length N.
In one move, you can pick i and j such that $A_i \neq A_j$ and set them both to $A_i \oplus A_j$.

Find a sequence of at most $20 \cdot N$ moves that results in a sorted array.

EXPLANATION:

The XOR operation is a bit of a red herring: there exists a solution that is completely independent of the replacement function.

A simpler case

First, let's deal with a simpler version of the problem, where N is always a power of two; say $N = 2^K$.

In this case, we can not only sort the array, we can make all the array elements equal using $K \cdot 2^{K-1}$ operations:

- First, recursively make the first and second halves of the array equal, since they're also power-of-two sized.
- Then, perform the operations $(1, N/2+1), (2, N/2+2), (3, N/2+3), \ldots, (N/2, N)$.
  Since the first half is equal and the second half is equal, the entire array is now equal.

It's easy to see that this uses $K \cdot 2^{K-1}$ moves: there are K levels of the recursion, and each one uses $N/2 = 2^{K-1}$ operations in total.

Solving for any N

When N isn't a power of 2, we can in fact use the above method as a subroutine.

First, perform $N/2$ operations to make A a palindrome: $(1, N), (2, N-1), (3, N-2), \ldots, (N/2, N/2+1)$

Now, let x be the largest power of 2 that's $\leq N$.

Let's first make the first x elements equal using the earlier method, and then make the last x elements equal.
Note that this gives us either a sorted array or a reverse-sorted array, with at most two distinct elements.

if it's sorted, we're done.
If it's reverse-sorted, note that our method for powers of 2 is symmetric with respect to reversing the array.
So, we can instead make the last x elements equal, *then* make the first x elements equal: this is guaranteed to give us a sorted array.

This way, we use the $K \cdot 2^{K-1}$ method twice, with a further $N/2$ operations.
$N \leq 10^5$ gives us $K \leq 16$, so the number of operations we use is bounded by $2 \cdot 16 \cdot N/2 + N/2 \leq 17N$, which is good enough.

TIME COMPLEXITY

$\mathrm{O}(N \log N)$ per test case.

PREREQUISITES:

Depth-first search

PROBLEM:

You have a tree with N vertices, rooted at 1. Initially each vertex has $A_i = 0$.
Process Q queries on it:

- Type 1: given u,
  - If u is a leaf set $A_u = 1$
  - Otherwise set A_u to be the bitwise AND of all descendants of u.
- Type 2: count the number of good edges in the tree.
  - An edge is good if the two components obtained on deleting it have the same bitwise AND.

EXPLANATION:

For a node u, let p_u denote the parent of u.

Notice that under the given operation, if $A_u = 0$ then $A_{p_u} = 0$ will definitely hold, since we can only turn a 0 into a 1 and this can only done in a bottom-up fashion starting from the leaves.

Now let's analyze what a good edge looks like.
Let's remove the edge between u and p_u (of course, $u > 1$); let C_1 denote the subtree of u and C_2 denote the rest of the tree. Let x_1 and x_2 be their respective bitwise ANDs; we'd like $x_1 = x_2$ to hold.

- If $A_u = 0$, then $x_1 = 0$. Our earlier observation tells us that $A_{p_u} = 0$, which means $x_2 = 0$ as well. So, this edge is definitely good.
- If $A_u = 1$, then every node in the subtree of u is 1, so $x_1 = 1$. For $x_2 = 1$ to hold, *every vertex* in the tree must then have a value of 1; equivalently, $A_1 = 1$ must hold.

So, we have the following conclusions:

- If $A_1 = 1$, then every edge is good
- Otherwise, the answer is the number of $u > 1$ such that $A_u = 0$.

All we need to do is maintain this information across updates.
Let ans denote the answer; initially $ans = N - 1$ since every edge is good.

When we update a vertex u:

- If A_u doesn't change, then nothing changes.
- If A_u turns from 0 to 1, then the edge between u and p_u is no longer good so ans decreases by 1.

To quickly check this condition, note that we have the following observation:
A_u turns from 0 to 1 if and only if $A_u = 0$ and $A_v = 1$ for *every child* v of u.

Iterating across all the children of u for every update would take too much time, since we might do it multiple times for a given u.
Instead, let's do the reverse: whenever A_u changes to 1, we propagate that information to its parent p_u.

That is, let's keep an array ch, where ch_u denotes the number of children of u that are 0.
Initially, ch_u equals the number of children of u, which can be computed using a DFS.

Now, when updating a vertex u,

- If $A_u = 1$ or $ch_u = 0$, do nothing
- Otherwise, A_u changes from 0 to 1. When this happens, simply subtract 1 from ch_{p_u}, since p_u now has one less child that is 0.

This allows us to process each update in $\mathrm{O}(1)$ time, leading to a solution in $\mathrm{O}(N + Q)$ overall.

TIME COMPLEXITY:

$\mathrm{O}(N + Q)$ per testcase.

PROBLEM:

You have an array A.
In one move, you can subtract 1 from A_i and A_{i+1}; indices modulo N.

Is it possible to make every element of A become 0?

EXPLANATION:

All indices mentioned below are cyclic modulo N and zero-based.
So, for example talking about A_{i-1} when $i = 0$ refers to A_{N-1} instead.

Let's deal with the cases of even N and odd N separately.

Even N

First, note that every move subtracts 1 each from an odd index and an even index.
So, surely the sum of values at even and odd indices should be equal; for the array to end up with zeros.
If this check fails, the answer is `No`, so let's assume it holds from now on.

Suppose we perform B_i moves on index i.
That is, we subtract 1 from the pair (A_i, A_{i+1}), B_i times.
In particular, the final value of A_i will be $A_i - B_i - B_{i-1}$.

Assuming that every element of A becomes 0 in the end, this gives us a system of equations in A_i and B_I.
Rearranging them a bit, it can be seen that

$$\begin{gathered}
A_1 - B_1 - B_0 = 0 \implies B_1 = A_1 - B_0 \\
A_2 - B_2 - B_1 = 0 \implies B_2 = A_2 - B_1 = A_2 - A_1 + B_0 \\
A_3 - B_3 - B_2 = 0 \implies B_3 = A_3 - A_2 + A_1 - B_0 \\
\vdots \\
B_i = (-1)^i B_0 + \sum_{j=1}^{i} (-1)^{i-j} A_j
\end{gathered}$$

Notice that fixing B_0 then fixes *every* other B_i, so we only really need to check if a value of B_0 exists such that every B_i becomes valid.
That is, we need to ensure that every $B_i \geq 0$, since it's the number of moves we make.

Putting that into the equations we have, we obtain the constraints

$$\begin{aligned}
B_0 &\leq \sum_{j=1}^{i} (-1)^{j+1} A_j, && \text{for odd } i \\
B_0 &\geq \sum_{j=1}^{i} (-1)^{j+1} A_j, && \text{for even } i
\end{aligned}$$

This gives us lower and upper bounds on the possible values of B_0.

Let the lower and upper bounds obtained this way be L and R.
Then, a valid solution exists if and only if:

- $L \leq R$

- $R \geq 0$

Which allows us to choose a non-negative value of B_0 in the range $[L, R]$, hence satisfying all the equations and inequalities.

Finding L and R can be done in $\mathrm{O}(N)$, since we're really just looking at alternating prefix sums of A.
Odd N

Define B_i as in the even case.

The system of equations we set up is also exactly the same.
In particular, notice that this time, we have

$$B_{N-1} = B_0 - A_1 + A_2 - A_3 + \ldots + A_{N-1}$$

We also have $A_0 - B_0 - B_{N-1} = 0$, giving us $B_{N-1} = A_0 - B_0$.

Notice that this uniquely determines B_0, in particular

$$2 \cdot B_0 = A_0 + A_1 - A_2 + A_3 - \ldots - A_{N-1}$$

B_0 has to be a non-negative integer, meaning the RHS has to be even and non-negative; if it isn't, the answer is immediately `No`.
Otherwise, compute B_0, after which all the other B_i are uniquely determined as well.

So, compute all the other B_i and check if they're all ≥ 0.

TIME COMPLEXITY

$\mathrm{O}(N)$ per testcase.

PROBLEM:

You have a binary string S of length N. You can either delete one character from S or flip every character of S. Find the minimum number of moves to obtain a string consisting of all zeros.

EXPLANATION:

First, notice that the order in which operations are performed doesn't really matter.
So, let's perform all flips before all deletions.

Now, note that it's never optimal to perform a flip operation more than once — flipping an even number of times is equivalent to not flipping at all; while flipping an odd number of times is equivalent to flipping once.

So, we flip either zero times or one time.

- If we flip zero times, our only option is to delete every 1 from the string.
- If we flip one time, we must delete every 1 from the flipped string. The number of these equals the number of zeros in the original string.

So, suppose S has c_0 zeros and c_1 ones. The answer is then $\min(c_1, 1 + c_0)$.

TIME COMPLEXITY:

$\mathrm{O}(N)$ per testcase.