PROBLEM:

You are given two integers N and K. Find number of ordered triplets (A, B, C) that satisfy the following conditions:

- $0 \leq A, B, C < 2^N$
- A, B and C are **distinct**
- $A \oplus B \oplus C = K$

Here, $\oplus$ denotes the **bitwise XOR operation**.

EXPLANATION:

Observation 1: Given any number A $(\leq 2^N)$, we can always select two numbers B and C such that their XOR is K.
This is because for each bit of A, we can set bits of B and C accordingly such that the final XOR of the three is equal to the bit of K.
Let's have bit representation of a number X be as:

$$X = X_N X_{N-1} ..... X_1$$

Now for K if at any position i (going from right to left), its bit would be K_i. Lets take two cases:

- K_i = 1:
  - For $A_i = 1$: To make final XOR as 1, (B_i, C_i) could be either $(1, 1)$ or $(0, 0)$
  - For $A_i = 0$: To make final XOR as 1, (B_i, C_i) could be either $(1, 0)$ or $(0, 1)$
- K_i = 0:
  - For $A_i = 1$: To make final XOR as 0, (B_i, C_i) could be either $(1, 0)$ or $(0, 1)$
  - For $A_i = 0$: To make final XOR as 0, (B_i, C_i) could be either $(1, 1)$ or $(0, 0)$

Observation 2: None of the numbers can be K since then the other two numbers would become equal to each other.

Thus now for selecting A we would have $2^N - 1$ options. As for B and C, for each bit position of A, we would have 2 ways of selecting bits of B and C. Among all the possible selections of B and C, there shall be 1 case where B is equal to A and 1 case where C will be equal to A. Thus after subtracting these cases we will have $2^N - 2$ ways of selecting B and C. Thus our final answer would be:

$$ans = (2^N - 1) \times (2^N - 2)$$

TIME COMPLEXITY:

$O(logN)$, for each test case.

PROBLEM:

Chef has an array A of length N.

In one operation, Chef can do the following:

- Select any non-empty subsequence $S = \{S_1, S_2, S_3, \ldots S_k\}$ of A (where the S_i are the **indices** of the array representing the chosen subsequence)
- Then, choose one index from this subsequence, say S_j
- Finally, either add A_{S_j} to every element of the subsequence, or subtract A_{S_j} from every element of the subsequence. Formally, Chef must choose to do exactly one of:
  - Set $A_{S_i} = A_{S_i} + A_{S_j}$ for every $1 \leq i \leq k$, or
  - Set $A_{S_i} = A_{S_i} - A_{S_j}$ for every $1 \leq i \leq k$

Chef's task is to make all the elements of the array equal within 100 operations.

Note that after each operation, $-10^{18} \leq A_i \leq 10^{18}$ should hold for every element of the array.

See output details for instructions regarding the output format.

EXPLANATION:

We can observe that none of the numbers in the array is greater than 2^{30} so what we can do is to go through ranges from $2^{29} - 2^{30}$, to $2^{28} - 2^{29}$ and so on and for each range say $2^{i-1} - 2^{i}$, select all the numbers that fall in this range as our set S and choose the minimum number, say X among them and subtract all the numbers in S by X. In each such step we reduce all the selected numbers to atleast half of its value so we can reduce all the numbers to 0, in less than 100 steps.

In order to further optimise this, instead of going through the ranges in order, in each step we can select the maximum element, say M of the array and select the range as $(\frac{M}{2}, M)$.

TIME COMPLEXITY:

$O(Nlog(M))$, where $M = max(A_i), 1 \leq i \leq N$

PROBLEM:

We are given an array B. Elements of B are formed from the elements of array A using the following operation

- B_i = (A_i + A_{i+1}) % 2 for 1 <= i <= N-1
- B_N = (A_1 + A_N) % 2
- B_i = 0 OR B_i = 1

Given the array B, can there exist an array A such that B can be formed from A?

EXPLANATION:

When do we get 0's in B? We will get 0s if both Ai and Ai+1 are even or both are odd. When 2 numbers are even or both numbers are odd, their sum is necessarily even.
When do we get 1's in B? We will get 1s if either A_i is even and A_{i+1} is odd or vice versa.

Implementation
Lets try and construct the array A from array B given the conditions above. Let the 1st element in array A be [1]
We can iterate across B from i = 1 to i = N - 1.
For each i in B from i = 1 to i = N - 1, if B_i = 0, then we append A_{i+1} = A_i
For each i in B from i = 1 to i = N - 1, if Bi not equal to 0, then we append A_{i+1} = (A_i + 1) basically if A_i is odd, we are appending an even number. If A_i is even, we are appending an odd number

Now, our array A is constructed with N elements and we need to check if the 1st and Nth element of A meet the condition which will create BN. We can do this by checking for the condition
If (A_1 + A_N) % 2 = B_N, then A is a valid array. Correspondingly, B then is also a valid array.
If (A_1 + A_N) % 2 is not equal to B_N, then A is invalid. Hence B cannot be constructed from A

TIME COMPLEXITY:

Time complexity is $O(N)$.

PREREQUISITES:

Dynamic programming

PROBLEM:

JJ has an array A and an integer X. At most once, he can choose a subsequence of A and add X to all its elements.

What is the maximum possible value of $\sum_{i=2}^{N}(A_i \oplus A_{i-1})$ that he can obtain?

EXPLANATION:

This task can be solved fairly easily with the help of dynamic programming.

Suppose we define a new array dp of length N, where dp_i is the maximum possible answer for the first i elements. Our aim is to compute dp_N.

Each element has two choices for it: it either remains A_i, or becomes $A_i + X$. This information can be encapsulated into the state for dynamic programming by simply holding two values for each index: $dp_{i,1}$ denotes the maximum answer for the first i elements if A_i isn't changed, and $dp_{i,2}$ denotes the maximum answer for the first i elements if A_i is changed to $A_i + X$.

All that remains are the base cases and transitions.

- The base case is $i = 1$, where we have $dp_{1,1} = dp_{1,2} = 0$
- To compute $dp_{i,1}$, we have 2 choices for the previous element: either it changed, or it did not.
  - If it did not change, we get the value $dp_{i-1,1} + (A_i \oplus A_{i-1})$
  - If it did change, we get the value $dp_{i-1,2} + (A_i \oplus (A_{i-1} + X))$
  - $dp_{i,1}$ is the maximum of these two values.
- Similarly, $dp_{i,2}$ is the maximum of $dp_{i-1,1} + ((A_i + X) \oplus A_{i-1})$ and $dp_{i-1,2} + ((A_i + X) \oplus (A_{i-1} + X))$.

These values can be computed iteratively, simply iterating i from 2 to N. The final answer is the maximum of $dp_{N,1}$ and $dp_{N,2}$.

TIME COMPLEXITY:

$O(N)$ per test case.

PREREQUISITES:

Diameter of a tree

PROBLEM:

You are given a tree on N vertices and Q queries, each of the form (u, v). For each query, find the largest integer k such that there exists a set of vertices $\{x_1, x_2, \ldots, x_k\}$ satisfying:

- $x_1 = u$
- $x_k = v$
- For each $1 \leq i < k$, $dist(x_i, x_{i+1}) = i$ where the distance between two vertices is the number of edges on the unique shortest path between them.

A further constraint is that u is never a leaf.

EXPLANATION:

As it turns out, the actual solution to a query is rather simple — proving that it works is the hard part.

First, consider things from the perspective of v.

- We need to end at v with a path of length $k-1$, so of course a path of this length from v must exist. Let M be the maximum length of a path that has v as one of its endpoints. Then, $M+1$ is an upper bound for the answer.
- Second, we make a total of $1 + 2 + \ldots + (k-1)$ steps from u to v, which equals $k(k-1)/2$. This number must definitely have the same parity as the length of the (unique) $u-v$ path in the tree.

This is in fact sufficient. Let M be as described above, and p be the parity of the $u-v$ path in the tree. The answer is then the largest integer k such that $k \leq M+1$ and $k(k-1)/2$ has the same parity as p.
If we are able to compute M, then finding k is trivial: since only parity matters, the answer is going to be one of $\{M+1, M, M-1\}$ so just find the largest of them that satisfies the parity condition.

Computing M is also fairly easy. It is well-known that in a tree, the longest path from a vertex has its other endpoint as one of the endpoints of the tree's diameter. So, use any standard algorithm (for example, use dfs/bfs twice) to compute the endpoints of a diameter of the tree, then compute the distance from these two endpoints to all the other vertices. This precomputation can be done in $O(N)$, after which each query is answered in $O(1)$.

Now comes the harder part: proving that the above solution is indeed correct. Note that the below proof depends specifically on the fact that u is not a leaf.

Proof

The below proof is from the author, and will be edited a bit to add more detail later.

Let k be the value we computed above, i.e, the largest integer that is $\leq M$ and $k(k-1)/2$ has the same parity as that of the $u-v$ path.

This k is an obvious upper bound on the answer since those two conditions are necessary. To prove that they are sufficient, we can explicitly construct an appropriate sequence.

Let's make the moves backwards, i.e, start from v then make steps of length $k, k-1, \ldots 1$ to reach u.

Define a function f, where $f(i,j,x)$ is true if it is possible to reach j from i with the first step being of length exactly x, and false otherwise.

Claim: If the following three conditions hold, $f(i,j,x)$ is true:

- $x(x+1)/2 \geq dist(i,j)$
- $x \leq dist(i,j)$
- $dist(i,j)$ and $x(x+1)/2$ have the same parity.

Proof

Let u denote the current vertex. Initially, $u = i$. Repeat the following strategy:

- If $dist(u,j) \leq x$, jump directly towards j
- Otherwise, jump directly towards i (and hence, away from j)
- Then, decrease x by 1 and continue the process

This process does the following:

- Since $x \leq dist(i,j)$, the first move is guaranteed to be a jump towards j. We then make some more jumps toward j.
- Consider the first moment we make a jump away from j. After this point, any jump we make of some length L will ensure that we are within a distance of $2L-1$ from j.
- In particular, when we make the jump with $L = 1$, we are within a distance of 1 from j. However, the parity condition mentioned above guarantees that we reach exactly j.

The second point above that we stay within distance $2L-1$ has some counterexamples if the current distance is ≤ 3 and j is a leaf. However, the input guarantees that u is not a leaf (and u is what we treat as j in this case), and so these counterexamples don't work.

Now for the actual construction: given u and v, find a vertex g such that $dist(g,v) = k$.
Then, from v, alternate jumping towards g and towards v till you reach a point where the distance from the current vertex to u is larger than the step size that needs to be made (recall that we are making moves in decreasing order of length, starting from v).

Such a moment always exists. Let the length of the current jump be L. It further holds that the distance from the current vertex to u is $\leq L(L+1)/2$ (again, there are some counterexamples when u is a leaf but they don't matter in this case).

The parity condition is also satisfied (since we ensured k was chosen to satisfy the parity condition initially).
Thus, all three conditions from our above claim are now true, so there exists a way to reach u with the final step.
This completes the construction.

TIME COMPLEXITY

$O(N+Q)$ per test case.

PREREQUISITES:

Strings, Binary Search

PROBLEM:

Chef likes to spend his summer vacation playing with string S. But he only likes the characters a and b to be part of his string.

Initially the string S is empty and Chef will build the string into his favourite alternate pattern in the following way.

- Add a once to string S.
- Add b twice to string S.
- Add a thrice to string S.
- and so on infinitely.

Now Chef will crop this infinitely large string to a length N and calculate the beauty of the string S.

The beauty of a string S is the sum of values of all substrings of S and the value of string S is the number of times the character changes from a to b or vice versa in the string.

Can you help Chef find the beauty of his string S of length N.
Since this calculated number can be large, output it modulo 1000000007.

EXPLANATION:

We can see if we take the first two characters then there would be 1 change and number of substrings that includes the first two characters are $N-1$.
Now we move on to the $2nd$ change that occurs at 3_{rd} position for first 4 characters of the string and number of substrings that includes this change would be $3 \times (N-3)$.
Similarly the number of substrings that includes the 3_{rd} change would be $6 \times (N-6)$.
So basically our answer would be something like:

$$\begin{aligned} ans &= 1 \times (N-1) + 3 \times (N-3).....k \times (N-k) \\ &= N \times (1+3+.....+k) - (1^2+3^2....+k^2) \end{aligned}$$

We can easily calculate k using binary search and pre-calculate the sum of the series to get out final answer.

TIME COMPLEXITY:

$O(log(N))$, for each test case.

PROBLEM:

Given a binary string S, you repeat the following operation K times:

- Simultaneously apply the following operation to every index i: if $S_i = 1$, then set $S_i = 0$. Further, if $S_{i-1} = 0$, set it to 1 and if $S_{i+1} = 0$, set it to 1.

Find the final state of the string.

EXPLANATION:

Simulate the first operation in $O(N)$ and reduce K by 1.

After this first move, every position behaves in a very predictable manner. Consider some index i such that $S_i = 1$. Then, this position will alternate between 1 and 0 in all future moves.

Proof

The first observation is that after a move is made, every 1 in S *must* be adjacent to a 0.

This can be proved by contradiction. Suppose some 1 is adjacent to only other 1's after a move.
Then, this index was initially 0, and all its neighbors were also initially 0. However, this leaves no way for this index to become 1 after a move is made!
So, every 1 must be adjacent to a 0.

Now the result follows immediately: once an index becomes 1, it has a 0 next to it.
After a move is made, the 1 becomes a 0 and the 0 becomes a 1.
After another move is made, both indices once again swap values.
This continues ad infinitum, regardless of what's going on in the rest of the string. This completes the proof.

This gives us a simple method of finding the final string:

- For each index i, find the first time it becomes 1. Denote this by $first_i$.
  - This can be done relatively easily: let the position of the closest 1 to the left of i be 1, and the closest 1 on the right be R. Then, the first time position i becomes 1 is $\min(i - L, R - i)$.
- Then, do the following:
  - If $first_i > K$, S_i remains 0
  - Otherwise, S_i is 1 if $K - first_i$ is even, and 0 otherwise.

Finding L and R for every index can be done in linear time with two scans over the array: one forward and one backward.

TIME COMPLEXITY

$O(N)$ per test case.

PREREQUISITES

Sorting

PROBLEM

There are N soldiers in an army and the i^{th} soldier has 3 parameters — *Attack points* : A_i, *Defense points* : $1000 - A_i$ and *Soldier type* : `ATTACK` or `DEFENSE`.

For the whole army:

- *Attack value* of the army is defined as the **sum** of *attack points* of all `ATTACK` type soldiers.
- *Defense value* of the army is defined as the **sum** of *defense points* of all `DEFENSE` type soldiers.
- *Rating* of the army is defined as the **product** of *Attack value* and *Defense value* of the army.

The task is to assign the *soldier type* to each of the soldiers to **maximize** the *rating* of the army and find out that maximum *rating*.

EXPLANATION

To solve this problem, let's first try solving a subproblem, which is to find the maximum achievable *rating* if you are given the count of soldiers of `ATTACK` and `DEFENSE` type.

Let's define *rating(r)* as the maximum possible *rating* that can be achieved, given that we can assign any r soldiers to `ATTACK` type and the remaining $n - r$ soldiers to `DEFENSE` type. The solution to this subproblem can be constructed as follows —

- Sort the *Attack points* array A in descending order.
- Assign first r soldiers to `ATTACK` type and remaining $n - r$ to `DEFENSE` type.
- So, *Attack points* of every soldier of `ATTACK` type are greater than or equal to attack points of every soldier of `DEFENSE` type.

Proof that the proposed approach gives the assignment with maximum rating

$A[i]$ are *Attack points* of soldier at index i and $D[i]$ are the *Defence points* of soldier at index i.

After following the proposed approach, we can divide the soldiers into 2 groups Atk and Def —

- $Atk_1, Atk_2, \ldots, Atk_r$ are indexes of the soldiers of `ATTACK` type.
- $Def_1, Def_2, \ldots, Def_{n-r}$ are indexes of the soldiers of `DEFENSE` type.
- $A[Atk_i] \geq A[Def_j]$ and $D[Atk_i] \leq D[Def_j]$ $(1 \leq i \leq r; 1 \leq j \leq n - r)$

Let's assume that the proposed approach is wrong and there is a better value of *rating(r)* that can be achieved by swapping types of soldiers Atk_i and Def_j $(1 \leq i \leq r; 1 \leq j \leq n - r)$. Let that the current *Attack value* and *Defense value* are $Atkval$ and $Defval$ respectively. Then the new values after swapping the types of soldiers would be —

- *New Attack value* $= Atkval - (A[Atk_i] - A[Def_j])$. As $A[Atk_i] \geq A[Def_j]$, *New Attack value* is lesser than or equal to $Atkval$.
- *New Defense value* $= Defval - (D[Def_j] - D[Atk_i]) =$. As $D[Atk_i] \leq D[Def_j]$, *New Defense value* is lesser than or equal to $Defval$.

As both *New Attack value* and *New Defense value* are lesser than or equal to $Atkval$ and $Defval$ respectively, the *New Rating* would also be lesser than or equal to the initial *rating*.

So, we have a contradiction as we couldn't get a better value of *rating(r)* and hence the approach that we proposed would always give the maximum value of *rating(r)*.

Let B be the array that we got by sorting *Attack points* array A in descending order, $Sum(r)$ be the sum of attack points of first r soldiers in array B and $TotSum$ be the total sum of attack points of all soldiers.

We can simplify the expression of *rating(r)* using the steps below —

- *Attack value* × *Defense value*
- $[B_1 + B_2 ... + B_r] \times [(1000 - B_{r+1}) + (1000 - B_{r+2}) ... + (1000 - B_n)]$
- $[Sum(r)] \times [(1000 * (n - r)) - (B_{r+1} + B_{r+2} ... + B_n)]$
- $[Sum(r)] \times [(1000 * (n - r)) - (TotSum - Sum(r))]$

Our final answer would be maximum value of *rating(r)* where r ranges from 0 to n. This can be implemented by iterating over array B and finding $Sum(r)$ for every r as $Sum(r-1) + B_r$ and putting all required values into the expression of *rating(r)* formulated above.

TIME COMPLEXITY

The overall time complexity is dependent on the sorting technique used. Everything except sorting can be done in $O(N)$. If we use fast sorting techniques, then an overall time complexity $O(Nlog(N))$ per test case can be achieved.

PROBLEM:

Chef has an array A of length N. In one operation, Chef can choose any element A_i and split it into two **positive** integers X and Y such that $X + Y = A_i$.
Note that the length of array increases by 1 after every operation.

Determine the **minimum** numbers of operations required by Chef to make **parity** of all the elements **same**.

It is guaranteed that parity of all the elements can be made equal after applying the above operation zero or more times.

EXPLANATION:

$Observation\ 1$: We can convert an even number, say x to two odd numbers by splitting into 1 and $x - 1$ but we cannot convert an odd number to two even numbers.

Thus if there is any odd number present in the array we will have to make parity of each element in the array as odd.

Therefore there can be two cases:
$Case\ 1$: All elements are even
In this case answer would be 0, since parity of each element is same, that is even.

$Case\ 2$: All elements are not even:
In this case, we will calculate number of even elements and our answer would be the number of even elements since we will apply operation on each even element to split it into two odd elements.

TIME COMPLEXITY:

$O(N)$, for each test case.

PROBLEM:

Chef has an array A $(1 \leq A_i \leq 10^5)$ of length N.

Let

$pref_i = A_1 + A_2 + \ldots A_i$

$suff_i = A_i + A_{i+1} + \ldots A_N$

Now Chef created another array B of length N such that $B_i = pref_i + suff_i$.

Chef lost the original array A and wants to recover it with the help of B. Help Chef to recover the array.

It is guaranteed in the input that there exists a valid array A for the given array B. In case of multiple valid arrays A, output any valid array.

EXPLANATION:

Hint

There exists a unique solution.

Solution

Let us have a look at the elements of array B.
$B_i = pref_i + suff_i = (A_1 + A_2 + ...A_i) + (A_i + A_{i+1} + ...A_N) = (A_1 + A_2 + ...A_N) + A_i$. Hence $B_i = SUM_A + A_i$, where SUM_A is the sum of the original array (A).

Now, let us have a look at the sum of the elements of the array B and use the above observation.
$SUM_B = B_1 + B_2 + ...B_N = (SUM_A + A_1) + (SUM_A + A_2) + ...(SUM_A + A_N)$.
Hence $SUM_B = (N + 1) \times SUM_A$.

So, we can find out SUM_A using SUM_B i.e. $SUM_A = SUM_B/(N + 1)$. After knowing SUM_A and using the first observation $(A_i = B_i - SUM_A)$, we can find all the elements of the original array (A), which is the required answer.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

Let A be a permutation of $\{1, 2, 3, \ldots, N\}$.

We define a function $f(A)$ as the **minimum** number of **increasing** subarrays into which the permutation A can be **partitioned**.
For example $f(\{5, 1, 4, 2, 3\}) = 3$, as the minimum number of increasing subarrays it can be partitioned into is 3. The partitions are: $\{5\}$, $\{1, 4\}$, and $\{2, 3\}$.

Chef has a permutation P of $\{1, 2, 3, \ldots, N\}$.
Chef wants to **sort** the permutation P. For that, he can choose any two indices (i, j) $(1 \leq i < j \leq N)$ and swap the elements P_i and P_j such that the value of $f(P)$ does **not increase** after the swap.

Help Chef sort the permutation in **at most** N^2 swaps.

EXPLANATION:

In order to sort the permutation P we will first take the first two increasing sequences of the array. In case the array has only one increasing sequence then that implies that the array is already sorted. Now after selecting the first two increasing sequences, we will sort them to make it a single increasing sequence. Again we will repeat the operation of selecting first two increasing sequences and sorting it till the whole array is sorted.

Now let's talk about how we will sort the first two increasing sequences, say L and R. We will select the largest element from R, say r that is smaller than the largest element of L, then using binary search we find the smallest element in L that is larger than r and swap the two. By repeatedly following this operation we can sort L and R into a single increasing sequence without increasing $f(P)$ and in atmost N^2 swaps.

TIME COMPLEXITY:

$O(N^2 log(N))$

PROBLEM:

You are given an array A. In one move, you can add 1 to any prefix of A. Find the minimum number of moves needed to make A a palindrome, or report that it is impossible to do so.

EXPLANATION:

Consider a second array B, where B_i denotes the number of times 1 was added to A_i during our operations. That is, the i-th element of the final array is $A_i + B_i$.

It's obvious that B is a decreasing array ($B_i \geq B_{i-1}$ for every $i > 1$), and the number of operations performed is exactly B_1. Our aim is to minimize B_1.

The final array should be a palindrome. So, if we consider some index i ($1 \leq i \leq N/2$), we want

$$A_i + B_i = A_{N+1-i} + B_{N+1-i}$$

Now, note that performing a move on a prefix larger than $N/2$ is pointless and can always be replaced by a shorter move that achieves the same result (do you see how?).

So, we can assume $B_i = 0$ for $i > N/2$. In particular, our equation above for a fixed index i changes to

$$A_i + B_i = A_{N+1-i}$$

Notice that this fixes the value of B_i, so we just need to check whether the B_i we obtain this way is a valid array or not.

Hence, the final solution is as follows:

- For each $1 \leq i \leq N/2$, compute $B_i = A_{N+1-i} - A_i$.
- If B is not a decreasing array, or any element of B is < 0, then it is impossible to make A a palindrome and the answer is -1.
- Otherwise, the answer is simply B_1.

TIME COMPLEXITY

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

PROBLEM

You are given an unsorted permutation P of size N. An operation is defined as:

Swap P_i and P_{i+K} for any i in the range $[1, N-K]$.

Find the maximum value of K, such that, the permutation P can be sorted by applying any finite number of operations.

EXPLANATION

Observation:

Assume P_i is not in i^{th} position where $1 \leq i \leq N$, to move P_i to i^{th} position, K should be such that it is a factor of the displacement i.e. $|P_i - i|$.

Proof

We can say that we would need to move P_i in one direction only, as moving in the other direction would cancel out the previous one. Let us say that we take x moves to move P_i to i^{th} position. In order for this to be possible, $x \times K = |P_i - i|$. Hence $x = |P_i - i|/K$ where x is a natural number. This implies K divides $|P_i - i|$ or in other words, K is a factor of $|P_i - i|$.

For example, in $[5, 2, 3, 4, 1]$, we need to move 5 from 1^{st} position to 5^{th} position, this can be done by setting $K = 4$ or $K = 2$ or $K = 1$.

Let us calculate all such displacement for which P_i is not in it's sorted position.

Now, to calculate the maximum value of K that can make the array sorted, we need to find K such that for all P_i not in i^{th} position, we can send it to i^{th} position by performing finite operations.

Hence, K must be a factor of all such displacement and should be maximum. So, we need to find maximum common factor or greatest common divisor of all such displacements. Therefore, the answer would be the gcd of all such displacements.

To calculate this, we will iterate over all N integers, calculate the gcd of $abs(P_i - i)$ where $1 \leq i \leq N$. This would be the required answer.

TIME COMPLEXITY

The time complexity is $O(N)$ per testcase.

PROBLEM:

There are N arrays of N length each. You are given the Mexes of all the arrays.
For each $0 \leq i \leq N-1$, Ashish wants to know the atleast amount of i in all N arrays and Kryptonix wants to know the atmost amount of i in all N arrays.

EXPLANATION:

Observation 1: Suppose MEX for a particular array A is $x(x > 0)$. Then all the numbers starting from 0 to $(x-1)$ would occur atleast once in this array.

Observation 2 : Suppose MEX for a particular array A is x, then any number except greater than x can occur atmost $(n-x)$ times in it.

Now in order to solve this problem we would to do some pre-computations. We would have a $freq$ variable to store frequency of each A_i and a $extra$ variable that would be sum of all positions where any number greater than $A_i(1 \leq i \leq n)$ can be put.

Keeping these in mind we would loop through the array and for particular position i, we would increment frequency of A_i by 1 and increment $extra$ by $n - A_i$.

Keeping these in mind, we would sort the array A and then loop from 0 to $n-1$. For a particular position say i, we first calculate the smallest j_{th} position in A such that $A_j > i$. Then all the arrays from j_{th} position to the end would have atleast 1 number as i, so $atleast[i] = n - j$. We can calculate $atmost[i]$ as:

$$atmost[i] = atleast[i] + extra - (freq[i] \times (n-i))$$

TIME COMPLEXITY:

$O(NlogN)$, for each test case.

PROBLEM:

You are given an integer N.

Find a permutation $P = [P_1, P_2, \ldots, P_N]$ of the integers $\{1, 2, \ldots, N\}$ such that sum of averages of all consecutive triplets is minimized, i.e.

$$\sum_{i=1}^{N-2} \frac{P_i + P_{i+1} + P_{i+2}}{3}$$

is minimized.

If multiple permutations are possible, print any of them.

EXPLANATION:

We are given the following sum to compute

$$\sum_{i=1}^{N-2} \frac{P_i + P_{i+1} + P_{i+2}}{3}$$

This can be simplified as:

$$\frac{P_1 + 2 \times P_2 + 3 \times (P_3 + P_4 ..... P_{n-2}) + 2 \times P_{n-1} + P_n}{3}$$

Since P_1 and P_n is not repeated and P_{n-1} and P_2 is repeated twice so it makes sense to assign maximum values to P_1 and P_n, followed by the next largest values to P_{n-1} and P_2. Rest of the values we can assign randomly since rest all of them are in multiples of 3. Thus we can have one of the possible permutations as follows:

$$N, N-2, 1, 2, 3 ..... N-3, N-1$$

TIME COMPLEXITY:

$O(N)$, for each test case.

PROBLEM:

A tuple of positive integers (a, b, c) is said to be a *bad tuple* if a, b, c are distinct, a divides b, and b divides c. For example, $(1, 2, 4)$ and $(7, 21, 42)$ are bad tuples while $(1, 2, 3)$ and $(2, 4, 4)$ are not.

Chef has the N integers from 1 to N with him. He wants to pick a **subset** of these numbers such that it doesn't contain a *bad tuple*. How many distinct subsets can he pick?

EXPLANATION:

WLOG, we can assume in a tuple (a, b, c), $a \leq b \leq c$,
Observation 1: The numbers greater than $\frac{N}{2}$ would not divide any other number and hence they can act only as c in the tuple.

Observation 2: Numbers greater than $\lfloor \frac{N}{3} \rfloor$ can only divide $2 \times themselves$ so they can only be present in tuples of form $(i, x, 2 \times x)$, where i is a divisor of x.

We will begin by finding all subsets not containing bad tuples consisting only of number from 1 to $\lfloor \frac{N}{3} \rfloor$. After finding such a tuple we will find pairs of numbers which have a divisor also present. Say numbers of form (i, y), where i divides y.

Now we will mark all multiples of such y greater than $\lfloor \frac{N}{3} \rfloor$ as bad as including them in the subset would lead to formation of a bad tuple. After excluding such numbers it is easy to see by **Observation 2** that the tuples would be of the form of $(i, x, 2 \times x)$, we can find all such x which would be paired with their twice. Hence finally we will have two types of numbers. One which have been paired and others which haven't.

The number which have been paired we can either include x or $2 \times x$ or none so we have three choices. For the numbers which haven't been paired we can either include the or exclude them so we have two choices. Hence the answer for this good subset would be:

$$2^{unpaired\ numbers} \times 3^{paired\ numbers}$$

The summation of all the answers of each good subset would be the final answer.

TIME COMPLEXITY:

$O(N \times 2^{\lfloor \frac{N}{3} \rfloor})$

PROBLEM:

Infinitely many people stand in a line, and some operations are performed:

- Type 1: A ball is given to the first person. Then, if someone has two balls, they drop one and pass the other to the person on their right. This continues as long as at least one person has two balls.
- Type 2: Each person with a ball passes it to the person on their left. If the first person has a ball, they drop it.

You are given the total number of operations N, and the K instances of time $(A_1, A_2, \ldots, A_K)$ when type 2 operations were performed. Find the total number of dropped balls.

EXPLANATION:

Let's treat the current state of the people as a binary string: the i-th character is 1 if person i currently holds a ball, and 0 otherwise. Let the number represented by this binary string be x.

Now, note that:

- A type 1 operation corresponds to adding 1 to x.
- A type 2 operation corresponds to dividing x by 2.

This allows to easily represent the current state of the people using only arithmetic operations.

We want to count the number of dropped balls at the end of the process. This is the same as the total number of balls introduced, minus the number of balls present at the end of the process.
These two quantities are not too hard to calculate:

- Note that each type 1 operation introduces exactly one new ball (and maybe drops some old balls, but we don't care about that). A type 2 operation doesn't introduce any new ball.
  So, the total number of balls is exactly $N - K$, i.e the number of type 1 operations.
- To compute the number of balls at the end, we can use the arithmetic operations above to quickly simulate the process.
  - Initially, $x = 0$.
  - Consider a type 2 operation at time A_i.
  - The previous type 2 operation was at A_{i-1} (for convenience, define $A_0 := 0$). So, exactly $A_i - A_{i-1} - 1$ type 1 operations were performed before this. Each of them adds 1 to x, so we increase x by $A_i - A_{i-1} - 1$. Now, divide x by 2 to simulate the type 2 operation at time A_i.
  - Finally, there are $N - A_K$ more type 1 operations at the end, so add this value to x.

This process lets us compute the final value of x in $\mathrm{O}(K)$. Once we know x, the number of balls currently with us is exactly the number of set bits in the binary representation of x, which is easy to compute.

Subtract this value from $N - K$ and we obtain our final answer.

TIME COMPLEXITY

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

PROBLEM:

You're given an array A of N integers. You need to find the minimum cost of creating another array B of N integers with the following properties

- $B_i \geq 0$ for each $1 \leq i \leq N$
- The GCD of adjacent elements of B is equal to 1, i.e, $\gcd(B_i, B_{i+1}) = 1$ for each $1 \leq i < N$

The cost of creating B is defined as follows:

$$\sum_{i=1}^{N} 2^{|A_i - B_i|}$$

Find the **minimum** possible cost to create the array B. Since the answer can be huge print it modulo $10^9 + 7$

Note: You need to minimize the value of total cost of creating the array B, and then print this minimum value modulo $10^9 + 7$. For example, suppose there is a way to create the required B with a cost of 500, and another way to do it with a cost of $10^9 + 7$ (which is $0 \pmod{10^9 + 7}$). The output in this case would be 500 and not 0.

EXPLANATION:

Observation 1: Let us take k as the $max(|A_i - B_i|),\ 1 \leq i \leq N$. We would note that k is small(less than 20).

Now using the above observation let us solve this problem using DP.
Let us define our dp $dp[i][j]$, where $1 \leq i \leq n$ and $-k \leq j \leq k$.
Here $dp[i][j]$ stores the minimum cost and denotes that we have processed till i_{th} index and the element at the i_{th} index is modified to $(A[i] + j)$. Now let us compute $dp[i][j]$.
Clearly $dp[i][j]$ depends on $dp[i-1][j']$, $where\ -k \leq j' \leq k$. As we are computing $dp[i][j]$, we know that the value of i_{th} index is $A[i] + j$, so $dp[i][j]$ depends on $dp[i-1][j']$ if and only if $gcd(A[i] + j, A[i-1] + j')$ is 1. In that case:

$$dp[i][j] = min(dp[i][j], dp[i-1][j'] + 2^{|j|})$$

So we compute this dp and our answer would be $min(dp[n][j]), where\ -k \leq j \leq k$

TIME COMPLEXITY:

$O(Nk^2)$, for each test case.

PREREQUISITES:

Breadth First Search, Depth First Search, Probability

PROBLEM:

Alice and Bob live in a country in which there are N cities (numbered from 1 to N) connected to each other using $N-1$ bidirectional roads in such a way that any two cities are reachable from each other.

This country has a special property: for each city in the country, there are **at most** 6 roads directly connected to it.

Alice and Bob both decide to go on a tour of the country, starting from city number 1. Alice uses Depth First traversal (DFS) order to travel through all the N cities while Bob uses Breadth First Traversal (BFS) order. At any city, if there is more than one possibility for the next city to be visited, then each option is equally likely to be chosen by both Alice and Bob.

The pseudocode of their respective traversals are as follows:

```
// Alice's DFS traversal
dfs(u):
    // Let C denote the set of children of u
    shuffle(C)
    // Let the current state of C be [C[1], C[2], ..., C[k]]
    for i from 1 to k:
        dfs(C[i])


// Bob's BFS traversal
queue Q
Q.push(1)
while Q is not empty:
    u = front(Q)
    pop(Q)
    // Let C denote the set of children of u
    shuffle(C)
    // Let the current state of C be [C[1], C[2], ..., C[k]]
    for i from 1 to k:
        Q.push(C[i])
```

Define the *index* of a city in some traversal order to be the number of distinct cities visited before that city during that traversal. Find the expected number of cities whose index in Bob's traversal is **strictly less** than its index in Alice's traversal.

It can be shown that this expected value can be expressed as a fraction $\frac{P}{Q}$, where P and Q are coprime integers, $P \geq 0$, $Q > 0$ and Q is coprime with 998244353. You should compute $(P \cdot Q^{-1}) \pmod{998\,244\,353}$, where Q^{-1} denotes the multiplicative inverse of Q modulo $998\,244\,353$.

EXPLANATION:

Let $D[i][j]$ be the probability that i_{th} city is at the j_{th} index in the DFS traversal, and $B[i][j]$ represent the same for BFS traversal. First of all, we will compute the matrices D and B. We can assume the given graph network as a tree rooted at node 1.

Computation of D :

Suppose we know $D[p][0], D[p][1], ..., D[p][N-1]$ for some node p. Also, let $(c_1, c_2, .., c_m)$ be the children of p. Now using the probabilities for p, we will compute the probabilities for each of its child.

There will be in total $m!$ ways in which Alice can travel the children of p. Therefore, probability of occurrence of each permutation is $\frac{1}{m!}$. Without loss of generality, let us assume a particular order, say $(c_1, c_2, .., c_m)$. If the index of p is suppose k in the DFS order then, the index of child c_i , $id[c_i]$ will be -

$id[c_i] = k + 1 + \sum_{j=1}^{i-1} SubtreeSize[c_j]$, where $SubtreeSize[j]$ is the size of subtree rooted at j.

So, we can update D as-

$D[c_i][id[c_i]]$ += $\frac{1}{m!} * D[p][k]$, for $(1 \leq i \leq m)$ and $(0 \leq k < N)$

Node 1 will always have index 0, so using this base case, we can compute the complete D matrix.
Computation of B :

Let p be a node in the tree and $(c_1, c_2, .., c_m)$ be its children. Suppose, we have computed B matrices for the subtree with each of it's children as root separately. Let them be named as $B_{c_1}, B_{c_2}, ..B_{c_m}$. Now, we will compute B matrix for the subtree rooted at p, i.e. B_p.
There will be in total $m!$ ways in which Bob can travel the children of p. Therefore, probability of occurrence of each permutation is $\frac{1}{m!}$.
Without loss of generality, Let us assume a particular order, say $(c_1, c_2, .., c_m)$. Now consider any node x at some level y in the subtree of node c_i. In the BFS traversal of the subtree rooted at p, index of x will always lie in the range L to R, where

$L = \sum_{j=0}^{j=y} level\_cnt[p][j] + \sum_{j=1}^{j=i-1} level\_cnt[c_j][y]$,
$R = L + level\_cnt[c_i][y]$

Here, $level\_cnt[a][b]$ denotes the count of nodes at a distance of b (or at level b) in the subtree of node a.
Also, in the BFS traversal of the subtree rooted at node c_i, node x will always lie in the range L' and R', where L' and R' are computed in similar way as L and R respectively for the subtree of c_i instead that of p. So, we can compute B_p in the folowing manner-

$B_p[x][L + j]$ += $\frac{1}{m!} * B_{c_i}[x][L' + j] \quad \forall\, j, 0 \leq j \leq R - L$

We need to update for all the nodes x in the subtree of node p for all possible $m!$ permutation of it's children.

Here the base case is when we are at the leaf node, in such case $B_p[p][0] = 1$ and $B_p[p][i] = 0$ for all $i > 0$.
And the final B matrix is same as B_1.

After Computation of B and D matrices, we can compute the required expected value (E) as follows-

$E = \sum_{i=1}^{i=N}$ (Probability that index of node i in BFS order is less than that in DFS order)

$E = \sum_{i=1}^{i=N} [\sum_{j=0}^{j=N-1} (B[i][j] * \sum_{k=j+1}^{k=N-1} D[i][k])]$

TIME COMPLEXITY:

$O(N^3)$ per test case

PREREQUISITES:

Prefix sums

PROBLEM:

You are given an array A and A queries on it. For each query, you are given two subarrays and an integer k. Find the number of pairs of elements, one from the first subarray and one from the second, such that their bitwise xor has the k-th bit set.

EXPLANATION:

Let's look at answering a single query (k, L_1, R_1, L_2, R_2) first: speeding it up to answer multiple queries can come later.

Suppose $A_i \oplus A_j$ has its k-th bit set. This is only possible when:

- A_i has its k-th bit set and A_j doesn't; or
- A_j has its k-th bit set and A_i doesn't

In particular, if take some A_i from $[L_1, R_1]$ with its k-th bit set, we can pair it with *any* A_j from $[L_2, R_2]$ whose k-th bit is unset.
Similarly, if take some A_i from $[L_1, R_1]$ with its k-th bit unset, we can pair it with *any* A_j from $[L_2, R_2]$ whose k-th bit is set.

This gives us a rather simple solution:

- Let S_1 be the number of elements in subarray $[L_1, R_1]$ that have the k-th bit set
- Let U_1 be the number of elements in subarray $[L_1, R_1]$ that have the k-th bit unset
- Let S_2 be the number of elements in subarray $[L_2, R_2]$ that have the k-th bit set
- Let U_2 be the number of elements in subarray $[L_2, R_2]$ that have the k-th bit unset

Then, the answer to this query is simply $S_1 \cdot U_2 + S_2 \cdot U_1$.

Computing S_1, S_2, U_1, U_2 is easy to do by looping across the subarrays, but that's not fast enough to answer multiple queries: we need something a bit faster.

Using prefix sums

Notice that, if k is fixed, we can treat each element of the array as being either 0 or 1 depending on whether it has the k-th bit set or not.

Then, the above variables simplify quite nicely:

- S_1 and S_2 are the number of ones in their respective ranges, or more specifically, just the sums of those ranges.
- U_1 and U_2 are the number of zeros in their respective ranges. Knowing S_1, S_2, and the lengths of the ranges is enough to compute these values (since $S_1 + U_1 = R_1 - L_1 + 1$ and $S_2 + U_2 = R_2 + L_2 - 1$).

Computing range sums quickly is a well-known application of prefix sums.
We need to maintain separate prefix sums for each k, but there are only 60 possible values of k anyway so this is not an issue.

That is, for each $0 \leq k < 60$, let $pref_{k,i}$ denote the number of elements in $[1, i]$ that have the k-th bit set.
Then,

- $S_1 = pref_{k,R_1} - pref_{k,L_1-1}$

- $S_2 = pref_{k,R_2} - pref_{k,L_2-1}$
- U_1 and U_2 can be computed as noted above.

This allows us to answer each query in $\mathrm{O}(1)$ time.

TIME COMPLEXITY

$\mathrm{O}(60 \cdot N + Q)$ per test case.

PROBLEM:

There are N cities in a row. The i-th city from the left has a sadness of A_i.

In an attempt to reduce the sadness of the cities, you can send **blimps** from the left of city 1 that move rightwards (i.e, a blimp crosses cities $1, 2, \ldots$ in order)

You are given two integers X and Y. For each blimp sent, you can make one of the following choices:

- Let the blimp fly over every city, in which case the sadness of **every city** will decrease by Y, or,
- Choose a city i $(1 \leq i \leq N)$, and shower confetti over city i. In this case, the sadness of cities $1, 2, \ldots, i-1$ will decrease by Y, the sadness of city i will decrease by X, and cities $i+1, \ldots, N$ see no change in sadness.

Find the **minimum** number of blimps needed such that, by making the above choice optimally for each blimp, you can ensure that no city is sad (i.e, in the end every city has sadness ≤ 0).

EXPLANATION:

$Case\ 1 : X \leq Y$: In this case, since X is always smaller than Y, so it won't make sense to shower confetti over any city so we would just let the blimp pass over each city and thus for each blimp the sadness of all cities would reduce by Y. Thus our answer would be

$$ans = \lceil \frac{A_{max}}{Y} \rceil$$
$$where\ A_{max} = max(A_i), 1 \leq i \leq N$$

$Case\ 2$: In this case it would be most efficient to show confetti to the farthest right city that has a positive value. This way all the cities before that would have their sadness reduced by Y and the last city would have its sadness reduced by X. Showering confetti before the last city with positive sadness won't make sense since then the cities after that won't have any sadness reduced.
In this case we would loop from end to start to calculate and for each say, the i_{th} city, we would calculate its current value, which would be the total number of blimps passed through it before the blimp started showering confetti on it. Let us assume total number of blimps passed as $total\_blimps$, and the sadness levels of the i_{th} city as $A[i]$

$$A[i] = max(0, A[i] - total\_blimps \times y)$$
$$total\_blimps = total\_blimps + \lceil \frac{A[i]}{X} \rceil$$

Our final answer would be $total\_blimps$ at the end of the loop.

TIME COMPLEXITY:

$O(N)$, for each test case.

PREREQUISITES:

Binary search or two-pointers

PROBLEM:

A binary string A is called *good* if it can be sorted by some sequence of adjacent swaps, such that any position is swapped with its right at most once.

Given a binary string S, how many of its substrings are good?

EXPLANATION:

The very first thing to do is to come up with a nice enough condition that tells us when a binary string is *good*.

The condition

Let A be a binary string. We'd like to sort A using some adjacent swaps.

Suppose $A_i = 1$ and $A_j = 0$, where $i < j$.
We definitely need to swap these at some point to sort the string.
This means that, no matter what, we must use the swaps at positions $i, i+1, i+2, \ldots, j-1$.

In particular, if we had to use some of these positions to swap a *different* pair of 1 and 0, we'd be in trouble.
This should immediately tell you that if A contains 1100 as a subsequence, it cannot be *good*.

It's now not hard to see that if A *doesn't* contain 1100 as a subsequence, it will always be *good*.

This gives us a nice reduction in what we want to compute: all we need to do now is compute the number of substrings that don't contain 1100 as a subsequence.
While this is easy to do in $O(N^2)$, that's obviously too slow.

Instead, we make one more observation. Let $S[L, R]$ denote the substring of S starting at L and ending at R.
Does knowing something about the goodness of $S[L, R]$ tell you something about $S[L, R-1]$ and/or $S[L, R+1]$?

Answer

If $S[L, R]$ is good, then so is $S[L, R-1]$.
Conversely, if $S[L, R]$ is not good, then neither is $S[L, R+1]$.

In particular, this tells us that if we fix L, the set of R such that $S[L, R]$ is good forms a continuous range starting at L.

So, let's fix L and try to find the *maximum* R such that $S[L, R]$ is good: then, we can add $R - L + 1$ to our answer since that's the number of good substrings starting at L.

To find R, we instead do the opposite: find the first time a 1100 subsequence forms, then end at the character right before that.
Finding the first time 1100 forms as a subsequence is not hard, and follows from the standard greedy algorithm to check whether one string is a subsequence of another:

- Let $i_1 \geq L$ be the first time we see a 1
- Let $i_2 > i_1$ be the first time we see a 1
- Let $i_3 > i_2$ be the first time we see a 0
- Let $i_4 > i_3$ be the first time we see a 0

Then, $R = i_4 - 1$.

Finding i_1, i_2, i_3, i_4 can each be done in $\mathrm{O}(\log N)$ if we have a sorted list of positions of the ones and zeros and simply binary search on this: for example, you can use `std::upper_bound` in C++ to simplify implementation.

So, we've found the optimal R for a fixed L in $\mathrm{O}(\log N)$. Do this for every L and add up the answers, giving us a solution in $\mathrm{O}(N \log N)$.

It is also possible to implement this in $\mathrm{O}(N)$ using two-pointers instead of binary search.

TIME COMPLEXITY

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

PREREQUISITES:

GCD

PROBLEM:

Given an integer N, construct an array A of length N such that:

- $1 \leq A_i \leq 10^{18}$ for all $(1 \leq i \leq N)$;
- For all $(1 \leq i \leq N)$, there exists a subsequence S of array A such that the length of the subsequence $2 \leq k \leq N$ and $gcd(S_1, S_2, \ldots, S_k) = i$.
  Note that $gcd(S_1, S_2, \ldots, S_k)$ denotes the gcd of all elements of the subsequence $S_1, S_2, \ldots, S_k$.

It can be proven that it is always possible to construct such A under given constraints. If there exist multiple such arrays, print any.

EXPLANATION:

Take four consecutive numbers, let the smallest one be a, $\rightarrow [a, a+1, a+2, a+3]$.
and on these indices assign values as $a \cdot (a+3), a \cdot (a+2), (a+1) \cdot (a+2), (a+1) \cdot (a+3)$. First two elements have a gcd of a (as $a+2$ and $a+3$ are coprime), similarly second and third element have a gcd of $a+2$, first and fourth have a gcd of $a+3$ and third and fourth have a gcd of $a+1$. Start assigning values from $i = N$ and keep moving towards 1 by assigning values in windows of size 4. In the end if 1 element remains assign it 1, if two elements remain assign them 1 and 2 otherwise assign 2, 3 and 6. In this way we have a subset of size 2 for each value of g from 1 to N such that the gcd of this subset is g.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Nim and the Sprague-Grundy theorem

PROBLEM:

Chef and Cook play a game on an array A, alternating turns with Chef starting first. In one move, the player may choose indices $1 \leq i < j \leq N$, subtract 1 from A_i, and add 1 to A_j. The player who cannot make a move loses.

Who wins with optimal play?

EXPLANATION:

A couple of observations allow us to reduce the given game to a standard game of nim, whose solution is well-known (and is linked above).

Reduction

For simplicity, let us say that we have N piles of stones, the i-th pile initially having A_i stones.
A move consists of moving a single stone from one pile to a later pile.

Looking at it this way, the movement of each stone is clearly independent, and so corresponds to a separate game.

Now, let's look at a single stone that is initially at position i. There are $N - i$ possible moves that can be made to its position — add 1, add 2, …, add $N - i$. As you might notice, this is exactly the same as having a pile of $N - i$ stones and playing a game of nim on it!

This gives us the desired reduction to nim: for each $1 \leq i \leq N$, we have A_i piles of $N - i$ stones, and then we simply find the solution to this nim game.

The above reduction to nim creates $A_1 + A_2 + \ldots + A_N$ piles in total, which is too large to store in memory. However, since checking who wins a game of nim depends only on the bitwise XOR-sum of all the piles' sizes, and we have the property $x \oplus x = 0$ for any x ($\oplus$ denotes bitwise XOR), we can do the following:

- If A_i is even, it contributes an even number of $N - i$ to the overall XOR-sum. This makes the overall contribution of this position 0.
- If A_i is odd, it contributes an odd number of $N - i$ to the overall XOR-sum. The overall contribution of this position is thus just $N - i$.

This gives us the final solution: compute the XOR of $N - i$ across all i such that A_i is odd; let this value be X. Chef wins if X is non-zero, and Cook wins otherwise.

TIME COMPLEXITY:

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

PROBLEM:

We are given a 10 x 10 grid with rows number 1 to 10 from top to bottom and columns numbered 1 to 10 from left to right. Each cell is defined by (r, c) where r is the row number and c is the column number. Chef can move from any (a, b) to any (c, d) within this grid as long as

- a not equal to c OR
- b not equal to d

EXPLANATION:

This is an implementation problem to check basic programming skills and logic.

- We know that we can move from ANY (a, b) to ANY (c, d) within the grid as long as the conditions are satisfied.
- Distance here is not a constraint - we can move from one corner of the grid to another in a single step.

What do the 2 points above tell us? We should be able to move to any other point in the grid -

- In 1 step if the point meets the condition such that (a not equal to c) and (b not equal to d)
- In 2 steps if the point meets the condition such that (a equal to c) or (b equal to d). In this case, in the 1st step - we will move to an intermediate point (e, f) such that (a not equal to e) or (b not equal to f). From that point, we will move to (c, d)

TIME COMPLEXITY:

Time complexity is $O(1)$.

PROBLEM:

You have a $N \times M$ grid, initially with all its cells white.
In one move, you can paint all the cells in one row or one column black.

Is it possible to have exactly K black cells after some sequence of moves?

EXPLANATION:

Notice that neither the order of moves nor which rows and columns were chosen matters at all: the only thing that matters is how many rows and how many columns were chosen.

So, suppose we chose x rows and y columns.
Then, the number of cells colored black is exactly $xM + yN - xy$.
We'd like to check if some choice of x and y allows for this to equal K.

Suppose we fix x.
Then,

$$xM + yN - xy = K \implies y(N - x) = K - xM$$

That is, y is uniquely fixed to be $\frac{K-xM}{N-x}$ once x is fixed.
Similarly, fixing y allows us to uniquely compute x.

However, x is the number of rows chosen, so it can be anything from 0 to N. Checking all of these would be too slow since N can be as large as 10^9.

To optimize this, we can make one simple observation: if we choose x rows and y columns, then we always color *at least* xy squares black.
This means that any valid solution must have $\min(x, y) \leq \sqrt{K}$.

Now we can simply loop over x and y to check if a valid solution exists! That is,

- For each x from 0 to $\sqrt{K}$, check if a valid y exists
- For each y from 0 to $\sqrt{K}$, check if a valid x exists

Note that when validating a pair (x, y) you must ensure that:

- x and y are integers
- $xM + yN - xy = K$
- $0 \leq x \leq N$ and $0 \leq y \leq M$

All of this can be done in $\mathrm{O}(1)$ for a given x and y, giving us a solution in $\mathrm{O}(\sqrt{K})$ overall.

TIME COMPLEXITY

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

PREREQUISITES:

Heaps

PROBLEM:

A 1-indexed array is called *positive* if every element of the array is greater than or equal to the index on which it lies. Formally, an array B of size M is called positive if $B_i \geq i$ for each $1 \leq i \leq M$.

For example, the arrays $[1], [2, 2], [3, 2, 4, 4]$ are positive while the arrays $[2, 1], [3, 1, 2]$ are not positive.

You are given an array A containing N integers. You want to distribute all elements of the array into some positive arrays. The elements of a positive array might not occur in the order they appear in the array A.

Find the **minimum** number of positive arrays that the elements of the array A can be divided into.

Please see the sample cases below for more clarity.

EXPLANATION:

Hint

It is always better to keep smaller elements before larger ones while forming a positive array because this will surely reduce the number of positive arrays required to distribute all the elements in A.
Hence, sorting will surely help!

Solution

From the hint it is clear that we should form positive arrays which have elements in an ascending order. If there are K elements in an existing positive array then the next element should be $\geq K + 1$.

We can greedily assign each element of A (in the ascending order) to some already existing positive array if it can accommodate the current element (if $A[i] \geq$ number of elements in the positive array $+1$) or in the other case we need to create another positive array.

We can maintain the information about all the existing positive arrays by storing the minimum required element for each of them.

If the current element $(A[i])$ can be part of any of the existing positive arrays i.e. it is $\geq$ the least among all the minimum required elements for the positive arrays then we can just update the minimum required element and there is no need to create another positive array. The least among all the minimum required elements can be easily obtained if we store all the minimum required elements in a priority queue and updating also becomes easy.

On the other hand, if the current element $(A[i])$ cannot become a part of any of the existing positive arrays then we create a new positive array with the current element $(A[i])$ being the first element in it i.e. add a new value (minimum requirement of 2) into the priority queue.

At the end of the iteration over A the size of the priority queue denotes the minimum number of positive arrays required i.e. our answer.

TIME COMPLEXITY:

$O(NlogN)$ for each test case.

PREREQUISITES:

Kruskal's algorithm to compute MST

PROBLEM:

Given N vertices where the i-th one has value A_i, you can draw an edge between u and v of length L if L is a submask of both A_u and A_v.
Find the minimum cost to connect all the cities, or claim it is impossible to do so.

EXPLANATION:

For the moment, let us suppose we can actually connect all N vertices. Let's make a couple of observations:

- The final graph is going to be a tree.
- Any edge L in the final graph will have its length be a power of 2, i.e, $L = 2^k$ for some $k \geq 0$.

Proof

The first point should be obvious: if we have a cycle, remove some edge on it to obtain strictly lower cost while preserving connectivity.

The second point is also not hard to see: if some length contains more than one set bit, removing any set bit in it will still keep it a valid edge while reducing cost.

This immediately gives us a solution, albeit a slow one:
Consider the (multi)graph G on N vertices, where for each (u, v, k) there is an edge between u and v of length 2^k if u and v *both* have the k-th bit set.
Our answer is then nothing but the weight of the minimum spanning tree of this graph.

However, this graph can have upto $30N^2$ edges, and computing them all is obviously impossible so we need to do better.

To optimize this, let's look at how Kruskal's MST algorithm would work on this graph:

- First, it'll consider all edges with weight 2^0
- Then, it'll consider all edges with weight 2^1
- Then, it'll consider all edges with weight 2^2

$\vdots$

Here's the nice thing: for a fixed k, the edges with weight 2^k have a rather special structure.

What?

Let $x_1 < x_2 < \ldots < x_m$ be all the vertices with the k-th bit set.
Then, the edges with weight 2^k are exactly *all* pairs of these m vertices.

Now, suppose we find $x_1, \ldots, x_m$ for a fixed k.
We don't need to consider *all* pairs of these m vertices: we simply need to keep enough edges to make them all connected. The easiest way to do this is to only consider the edges $(x_1, x_2), (x_2, x_3), \ldots, (x_{m-1}, x_m)$.

Notice that doing this immediately brings us down to $< N$ edges per bit, for a total of $< 30 \cdot N$ edges in total.
This is small enough that we can simply run a MST algorithm on these edges directly and output the answer.

Note that if the final graph we obtain is not connected, the answer is -1 since there's no way to connect it.

PROBLEM:

Given an array A of $2N$ integers, can it be split into two arrays of length N, each containing distinct elements?

EXPLANATION:

Suppose we split A into B and C, each containing distinct elements.

Consider some element x. x can appear at most once in B and at most once in C.
Of course, this means x can appear at most twice in A.

This applies to any integer, so *every* x must appear at most twice in A.

It's not hard to see that this condition is also sufficient, i.e, splitting into B and C is possible *if and only if* no integer appears 3 or more times in A.

Proof

If something appears 3 or more times, splitting is obviously impossible.

Suppose everything appears ≤ 2 times.
For every element that appears twice, put one copy in B and one copy in C.
There are at most N such elements, so B and C both have length $\leq N$ now.

The remaining elements can be distributed in any way to make B and C reach length N, since they will never break distinctness.

Checking this condition is fairly easy, and the constraints even allow for it to be done in $\mathrm{O}(N^2)$:

- Fix an index i $(1 \leq i \leq 2N)$
- Run a loop over A to count the number of times A_i appears in A.
- If this count is ≤ 2 for every i, the answer is `Yes`. Otherwise, the answer is `No`.

PREREQUISITES:

Segment trees with lazy propagation

PROBLEM:

Given N points (X_i, Y_i) on the plane, where $X_i < X_{i+1}$, support the following queries and updates:

- Given `L R K B`, set $Y_i := KX_i + B$ for each $L \le i \le R$
- Given `L R`, report whether the points in $[L, R]$ are collinear

EXPLANATION:

First, let's look at the collinear query. It can be rephrased in terms of slopes as follows:

- Define S_i to be the slope between the i-th and $(i+1)$-th point. That is, $S_i = \frac{Y_{i+1}-Y_i}{X_{i+1}-X_i}$.
- Then, the query $[L, R]$ really just asks if $S_L = S_{L+1} = S_{L+2} = \ldots = S_{R-1}$.

Computing the initial values of S_i is easy, so we just need to figure out how to maintain them across the given updates.

Let's break down what an update `L R K B` looks like in terms of the S_i.

- First, for each $L \le i < R$, we essentially just set $S_i = K$.
- The only other points that change value are S_{L-1} and S_R (if they exist).
- Finding out what they change to can be done if we are able to query for their respective y-coordinates. So, we also need to maintain the y-coordinates of the points across queries.

Note that the y-coordinate of a point depends solely on the last (K, B) update applied to it, so we can simply maintain the (K, B) values for each point. Each update is then just a range set update, which can be done using a segment tree.

From our discussion above, we see that maintaining the S_i values needs a data structure that can:

- Set a value of S_i to a range
- Set a value of S_i to a point (which can just be subsumed into the first point, treating a point as a length 1 range)
- Query for whether a range has all equal elements

The last part can be done by, for example, computing the maximum and minimum values in a range, and checking if they're equal.
So, we need a data structure that supports range set updates, and range min/max queries. Once again, this is just a segment tree with lazy propagation.

In order to not run into precision issues, it is recommended to maintain the S_i values as pairs of (numerator, denominator) instead of directly as floating-point numbers.

TIME COMPLEXITY

$O(N + Q \log N)$.

PROBLEM:

Given integers N and K, construct a circular array of size N such that every subarray of size K contains distinct elements, while minimizing the maximum element.

EXPLANATION:

Obviously, we need at least K colors.

A natural-seeming starting point is to then attempt to repeat these K colors, to form $[1, 2, 3, \ldots, K, 1, 2, \ldots, K, 1, 2, \ldots]$.

This will allow you to form $\lfloor \frac{N}{K} \rfloor$ 'blocks' of $[1, 2, \ldots, K]$, and will leave the last few elements uncolored — specifically, the last $N\%K$ elements will be uncolored.

Using $N\%K$ more colors gives us a valid array, of course, and we've used $K + N\%K$ colors. Can we do any better?

Turns out, we can!
The idea is not too hard: as before, let's place $\lfloor \frac{N}{K} \rfloor$ blocks of $[1, 2, \ldots, K]$.
This leaves us with $N\%K$ uncolored elements. Instead of placing them all at the end of the array, let's instead distribute them equally (as much as possible) to the ends of all the blocks.

For example, if there are 21 blocks and 118 uncolored elements, give 6 elements each to the first 13 blocks and 5 elements each to the remaining ones.
In particular, each block will receive either $\lceil \frac{N\%K}{B} \rceil$ or $\lfloor \frac{N\%K}{B} \rfloor$ uncolored elements, where B is the number of blocks.

Now notice that the resulting array can be colored quite easily using $\lceil \frac{N\%K}{B} \rceil$ extra colors, giving us a solution using $K + \lceil \frac{N\%K}{B} \rceil$ colors.
Constructing this coloring is fairly straightforward: as noted above, each block of uncolored elements has either size y or $y - 1$, where $y = \lceil \frac{N\%K}{B} \rceil$. So,

- Color a block of size y with colors $K + 1, K + 2, \ldots, K + y$
- Color a block of size $y - 1$ with colors $K + 1, K + 2, \ldots, K + y - 1$

That is, the final coloring will look like $[1, 2, 3, \ldots, K, K + 1, \ldots, K + y, 1, 2, \ldots, K + y, 1, \ldots, K + y, \ldots, 1, 2, \ldots, K + y - 1, 1, 2, \ldots, K + y - 1]$

It turns out that this is also optimal.

Proof

Suppose we are able to use C colors to color the array.
Let M be the maximum frequency of one of the colors; w.l.o.g say color 1.

Then, obviously, it must hold that $M \cdot C \geq N$.
Further, there are at least $K-1$ other elements between any two occurrences of 1.
This gives us a minimum of $M \cdot (K-1) + M = M \cdot K$ elements in the array, i.e, $M \cdot K \leq N$.

This tells us that $M \leq \lfloor \frac{N}{K} \rfloor$.

Now, recall that we have $M \cdot C \geq N$. We'd like to minimize C, so of course M should be as large as possible.
In other words, we choose $M = \lfloor \frac{N}{K} \rfloor$.

This tells us that

$$C \geq \lceil \frac{N}{\lfloor \frac{N}{K} \rfloor} \rceil$$

must hold.

The smallest C that satisfies this equation is indeed $K + \lceil \frac{N\%K}{\lfloor \frac{N}{K} \rfloor} \rceil$: this can be verified algebraically.

Of course, to solve the problem you don't need to solve this algebraically: you can, for example, run a loop on C or binary search to find the first time $C \cdot \lfloor \frac{N}{K} \rfloor \geq N$.

TIME COMPLEXITY

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

CODE:

Setter's code (C++)
Tester's code (C++)
Editorialist's code (Python)

Quote

nitinkumar1238 **Nov '22**

This was a really very nice problem.
I somehow managed to get that observation in the contest and solved it, but it was quite hard.

riyad000 **Nov '22**

for the input :
1
23 4
my code gives output as:
5
1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3
which is wrong. but my code got accepted.

bakru_k78 **Nov '22**

Simple Explanation

```
#include <bits/stdc++.h>
using namespace std;

#define int     long long int

void solve(){
    int n,k;
    cin>>n>>k;

    int noOfSubset = n/k;
    int leftOutElement = n%k;

    k += ceil(leftOutElement/(noOfSubset*1.0));

    vector<int> v(k), ans(n);
    for (int i = 0; i < k; ++i){ v[i] = i + 1; }

    cout<<k<<endl;
```

```
    for (int i = 0; i < n; ++i){
        ans[i] = v[i%k];
    }

    for(auto i : ans)
        cout<<i<<" ";
    cout<<endl;
}

signed main() {
    int t=1;
    cin>>t;
    while(t--) solve();
    return 0;
}
```

There is only problem when there are remain element left out in our array to be output. first if remainder of `n%k == 0` then we can simply print our answer.
But when the remainder remain then , calculate the number of subset and try to assign the remaining element to the no of subset . add this value to our k value. just print the answer.
Please to like this post.

---

blown_brains **Nov '22**

I was trying a binary search approach on number of colors. In order to validate whether x number of colors are possible or not, I tired a greedy coloring solution. This didn't pass the testcases at all! I spent huge time on this question. 😢

PROBLEM:

You have X, Y, Z units of the three primary colors. Two primary colors can be combined to make one secondary color.
What is the maximum number of distinct colors you can have?

EXPLANATION

There are a couple of ways to solve this problem: greedy/casework and bruteforce.

The bruteforce solution is simpler to think about and will be explained here.

It is enough to create either 0 or 1 drop of each secondary color, any more is pointless.
This gives us 8 possibilities for which secondary colors are created.
Simply try all 8 possibilities and take the best answer among them.
There are a few ways to implement this, perhaps the easiest is to use bitmasks. You can look at the editorialist's code for this.

If you tried a greedy solution and got WA, you likely failed on a testcase like

```
3
2 2 3
2 3 2
3 2 2
```

Note that they should all have the same answer (5).
Your greedy will probably work if you sort the input in descending order first.

TIME COMPLEXITY

$O(1)$ per test case.

PROBLEM:

You are given an array C of N distinct integers, that is the result of a merge (as in mergesort) operation between two arrays A and B. Find any two appropriate arrays A and B, or claim that they don't exist.

EXPLANATION:

One simple construction is as follows:

- Let $M = \max(C)$, and let i be the position of M in C, i.e, $C_i = M$.
- Set $A = [C_1, C_2, \ldots, C_{i-1}]$ and $B = [C_i, C_{i+1}, \ldots, C_N]$.
- If $i = 1$, then A is empty and we report -1 instead.

Why does this work?

- Suppose $i > 1$, so M is not the first element.
  - M is the first element of B, and every element of A is less then M.
  - So, all of A will be pushed into C first, and then since A is now empty, all of B will be pushed in, which is what we want.
- Suppose $i = 1$, so M is the first element.
  - Suppose a solution existed. Then, M must be the first element of either A or B.
  - Either way, all elements of the other array are less than M, and so will be pushed into C first.
  - However, since $C_1 = M$, this can only happen when one of the arrays is empty, which is not allowed.
  - So, no solution exists for this case.

TIME COMPLEXITY

$O(N)$ per test case.

PROBLEM:

There are N points lying on the coordinate axes.
In one second, you can draw some segments joining pairs of these points, as long as no two segments intersect non-trivially.

How many seconds do you need to draw every segment?

EXPLANATION:

Since all the points lie on the axes. every line segment we draw will be one of two types:

- Either it will lie completely on one of the coordinate axes; or
- It will lie entirely in one of the four quadrants.

Now, note that:

- Two line segments in different quadrants can never intersect nontrivially
- A line segment on an axis and another one in a quadrant can never intersect nontrivially

This gives us an idea to solve the problem: separately find the minimum time required to draw line segments in each quadrant, and along the axes. The final answer is the maximum of these times.
Let's look at them separately.

Quadrants

Let's consider the first quadrant, i.e, points (x, y) with $x, y > 0$.

Suppose there are k_1 points on the positive y-axis and k_2 points on the positive x-axis.

Then, it can be seen that you need $\min(k_1, k_2)$ seconds to draw all pairs of segments between them.

Proof

Let $y_1 < y_2 \ldots < y_{k_1}$ and $x_1 < x_2 < \ldots < x_{k_2}$ be the coordinates.

Consider what happens when we draw the segment joining $(0, y_{k_1})$ and $(x_1, 0)$.
It is now impossible for us to draw *any* segment from $(0, y_s)$ to $(x_t, 0)$ if $s < k_1$ and $t > 0$.

So, the best we can do is join y_{k_1} to all the x-points, and x_1 to all the y-points in this second.
Now we've drawn all segments from y_{k_1} and x_1, so we ignore them and continue on.
It's easy to see that this process will take exactly $\min(k_1, k_2)$ seconds to draw all pairs of segments since that's when one side is exhausted, and it's impossible to do any better.

So, just knowing the number of points on the axes for each quadrant will allow us to compute the answer for each one.

Axes

Segments on the axes are a bit trickier to handle.
There are two types of segments: ones lying on the x-axis, and ones lying on the y-axis.
Note that this time, there *can* be intersection between these different types of segments: but this intersection can only happen at the origin.

First, let's look at segments that cross the origin.

Notice that any step can contain at most one segment crossing the origin along the x-axis and at most one segment crossing the origin along the y-axis; but it also can't contain both of these.

In other words, each step can contain at most one segment crossing the origin. So, let's compute the total number of these.

Suppose there are c_1 segments crossing the origin along the x-axis and c_2 along the y-axis. The answer is at least $c_1 + c_2$.
Computing c_1 and c_2 is easy: if there are p points with **positive** x-coordinate, and n with **negative**, then $c_1 = p \cdot n$.
c_2 can be computed similarly.

Now, let's look at all segments on the axes.

Suppose there are m points on the x-axis. Then, we need a minimum of $\lfloor \frac{m}{2} \rfloor \cdot \lceil \frac{m}{2} \rceil$ seconds to draw all pairs of segments between them.

Proof

For convenience, let $m = 2k + 1$.
Let the points be $x_1 < x_2 < \ldots < x_m$.

Consider x_{k+1}.

- There are k points to its left and to its right.
- Any segment joining one point from the left and one point from the right needs its own second to be drawn, giving us k^2 seconds at least.
- Finally, we also need to draw segments from x_{k+1} itself. This can be done in k seconds by spreading outwards in both directions from x_{k+1}.
- Thus, we need $k \cdot (k + 1)$ seconds at minimum, which is exactly $\lfloor \frac{m}{2} \rfloor \cdot \lceil \frac{m}{2} \rceil$.

When $m = 2k$, a similar analysis will give you $k \cdot (k - 1) + k = k^2$ seconds, which once again matches that formula.

So, compute this value for the x-axis and y-axis separately, and take the maximum of both.

Note that while the latter computation does include segments that cross the origin, it doesn't matter: the value we computed is a clear lower bound for the answer, and it is also achievable.
Whenever we choose an origin-crossing x-segment, we can combine it with non-origin-crossing y-segments and vice-versa.
If non-crossing segments on one side run out, then we are limited by $c_1 + c_2$ from above anyway, so it causes no issues.

When we choose a y-segment that cross the origin, we can

The final answer is then the maximum value among everything computed above.

The only thing we did was count the number of points on the positive/negative x/y axes, which can be done in $\mathrm{O}(N)$.

TIME COMPLEXITY

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

PROBLEM:

Anton loves creating strings!

Anton now wants to create a string S following some specific rules. They are as follows:

Initially, S is empty. Then, Anton can perform two types of operations on S:

1. Choose a lowercase Latin character (an element of $\{a, b, c, \ldots, z\}$) and append it to S. For example, if currently $S = \text{clap}$, Anton can turn it into one of $\{\text{clapa}, \text{clapb}, \ldots, \text{clapz}\}$.
2. Append a copy of S to itself. For example, if currently $S = \text{clap}$, Anton can turn it into clapclap.

However, Anton doesn't want to perform operation 1 twice in a row.

You are given a string A consisting of the lowercase Latin alphabet. Is it possible for Anton to create A using his operations any number of times?

EXPLANATION:

Observation: The operation of first type can only be applied when the length of the string is even.
First type of operation can be applied when the string is empty or immediately after an operation of second type has been performed on the string. After every operation of second type the length of the resulting string is even ($2\cdot$ length of original string). Therefore the operation of first type can only be applied when the length of the string is even and finally resulting in an odd length string.
This means if the length of the string is even then the last operation performed on it has to be of second type otherwise if the length of the string is odd the last operation performed on it has to be of first type.

Start with the given string and keep on moving until the string becomes empty:

- If the size of the present string is odd, simply remove the last character of the string (operation of first type).
- If the size of the present string is even, Check whether this string is the copy of two equal strings(operation of second type) and reduce the size of the string by half. If the present string can't be obtained by a single operation of second type then stop iterating.

If the string is empty at the end then it can be constructed using these operations otherwise it cannot be constructed using these operations.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Dynamic programming, **next greater element**

PROBLEM:

An array is said to be good if the prefix upto its maximum element is sorted.

Given a permutation P of length N, find the number of ways to partition it into good subarrays.

EXPLANATION:

This is a counting problem, and most of the time that means the solution is going to involve either combinatorics or dynamic programming.
Combinatorics doesn't seem to be very helpful here, so let's try DP.

Let $P[L:R]$ denote the subarray of P starting at L and ending at R.

Let's try to make a couple of observations. Consider a subarray $P[L:R]$. Then,

- If $L = R$, then $P[L:R]$ is a good subarray.
- If $P[L:R]$ is a good subarray and $L < R$, then $P[L:R-1]$ is also a good subarray.
- If $P[L:R]$ is not a good subarray, then $P[L:R+1]$ is also not a good subarray.

All three facts should be fairly obvious to see.
Notice that this gives us a useful piece of information: if we fix the left endpoint L, then the set of R such that $P[L:R]$ is a good subarray form a contiguous range starting at L.
That is, the good subarrays starting at L are $P[L:L], P[L:L+1], P[L:L+2], \ldots, P[L:L+K]$ for some $K \geq 0$.

This immediately gives us an idea for a (slow) dynamic programming solution.

Initial idea

Let dp_i denote the number of ways to partition the suffix starting from i into good subarrays. Our final answer is dp_1, and the base case is $dp_N = 1$.

By fixing the length of the good subarray starting at i, it's easy to see that our transitions are simply

$$dp_i = dp_{i+1} + dp_{i+2} + \ldots + dp_{j+1}$$

where j is the largest integer such that $P[i:j]$ is good.

However, there are two problems with this solution:

- The first issue is that we need to find the right endpoint j (and find it quickly, at that)
- The second is that this is too slow. Each index can have a potentially $O(N)$ transition, in which case this approach has a runtime of $O(N^2)$.

Let's fix the problems individually.

Speeding up the DP

The speed issue is easy to fix using suffix sums.
Suppose we maintained the suffix sum array suf, where $suf_i = dp_i + dp_{i+1} + \ldots + dp_N$.
Then,

- $dp_i = dp_{i+1} + dp_{i+2} + \ldots + dp_{j+1} = suf_{i+1} - suf_{j+2}$
- $suf_i = suf_{i+1} + dp_i$

So, suf can be computed as we keep computing dp, and this allows us to optimize transitions to $\mathrm{O}(1)$.

Finding the right endpoint

Suppose you know the value of j for a left endpoint i. Can you find its value for left endpoint $i-1$?
It turns out, we can! Here's how:

- Suppose $P_{i-1} < P_i$. Then, the exact same value of j works for $i-1$ as well.
- Otherwise, $P_{i-1} > P_i$. In this case, let $x > i-1$ be the smallest integer such that $P_x > P_{i-1}$. Then, we choose $j = x-1$.

Proof

Both cases are not hard to prove.

- If $P_{i-1} < P_i$, and good subarray starting at $i-1$ can be turned into a good subarray starting at i by deleting the first element. Conversely, any good subarray starting at i still remains good when we insert P_{i-1} in the beginning. This implies that they have the same value of j.
- If $P_{i-1} > P_i$, the only sorted prefix is the one with length 1. So, the subarray starting here can only be good if P_{i-1} itself is the maximum element. So, the instant we hit something larger than P_{i-1}, the subarray is no longer good.

Dealing with the first case is trivial. However, dealing with the second needs us to find, for a given element, the first element to its right that is greater than it.

In fact, it is possible to find this next greater element for *every* index of the array in $\mathrm{O}(N)$, using a stack. The method of doing this is linked in the prerequisites.

Putting together the computation to find the right endpoint for each index, along with the DP speedup using suffix sums, gives us a solution in $\mathrm{O}(N)$.

TIME COMPLEXITY

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

PREREQUISITES

XOR, Graph, Combinatorics

PROBLEM

You are given an array A of N integers, initially all zero.

There are Q types of operations that you are allowed to perform. Each operation is described by two integers l_i and r_i $(1 \leq l_i \leq r_i \leq N)$ and is as follows:

- Set $A_j = A_j \oplus 1$ for each j such that $l_i \leq j \leq r_i$.

Here $\oplus$ represents the **Bitwise XOR** operator.

Your task is to find the number of distinct arrays which can be obtained by applying some subset of these operations (possibly none, possibly all). Since the answer may be huge, output it modulo 998244353.

EXPLANATION

Prepend $A_0 = 0$ and $A_{N+1} = 0$ to the array A.
Let us construct another array B of length $N + 1$ such that $B_i = A_i \oplus A_{i+1}$.
We can see that after applying any query $[L, R]$ exactly 2 values in the array B (B_{L-1} and B_R) will change. So for each query $[L, R]$ we will add an edge between the nodes $L - 1$ and R as values of both B_{L-1} and B_R will get changed after this operation.

There might be more than one components in the graph. It is easy to see that each component is different since the nodes concerned will be disjoint. So we will compute answer for all the components individually and multiply them to obtain the final answer.

Calculating Answer for one component

Consider a cycle in the component. If we apply operation on all the edges of a cycle then it will reset the values of all the nodes involved in the cycle. So we can remove the edges involved in the cycle and obtain a spanning tree. Now we can see the each edge is independent (i.e. changing the state of any edge will give us a new array). Let the number of nodes in the component be X. So there will be total $X - 1$ edges (Since we have obtained a tree). So the answer for this component will be 2^{X-1}. We can multiply the answers for each components to get our final answer.

TIME COMPLEXITY

$O(N + Q)$ for every test case

PREREQUISITES:

Derangement, next_permutation or random shuffle

PROBLEM:

You are given a permutation P of length N. A permutation of length N is an array where every element from 1 to N occurs exactly once.

You must perform some operations on the array to make it sorted in increasing order. In one operation, you must:

- Select two indices L and R $(1 \leq L < R \leq N)$
- Completely *dearrange* the subarray $P_L, P_{L+1}, \ldots P_R$

A dearrangement of an array A is any permutation B of A for which $B_i \neq A_i$ for all i.

For example, consider the array $A = [2, 1, 3, 4]$. Some examples of dearrangements of A are $[1, 2, 4, 3]$, $[3, 2, 4, 1]$ and $[4, 3, 2, 1]$. $[3, 5, 2, 1]$ is not a valid dearrangement since it is not a permutation of A. $[1, 2, 3, 4]$ is not a valid dearrangement either since $B_3 = A_3$ and $B_4 = A_4$.

Find the minimum number of operations required to sort the array in increasing order. It is guaranteed that the given permutation can be sorted in atmost 1000 operations.

HINT

The answer is always less than or equal to 2

EXPLANATION:

If the permutation P is already sorted the answer is 0. Let us look at the case when we can sort the permutation by just one move. If there exists a subarray $P[L, R]$ such that for each index i in the prefix $P[1, L-1]$, $P_i = i$ and for each index j in the suffix $P[R+1, N]$, $P_j = j$ and for all index k in the subarray $P[L, R]$, $P_k ! = k$. Then we can derange the subarray $P[L, R]$ to achieve the sorted premutation. Otherwise we can always do just 2 operations to obtain the sorted permutation.

By Brute Force we can check that for all permutations of size 4 or greater there exists a derangement such that for no index i, $P_i = i$ in the resulting derangement. Now, apply the operation on the whole permutation P. To obtain the derangement of P such that for no index i $P_i = i$ in the derangement we break the array into size 4 consecutive subarrays starting from the index 1. Merge the last subarray to the second last subarray if it has a size less than 4. Now iterate over all permutations possible of the elements of these 4 size subarrays and check whether the permutation is a derangement as well as for no index i in the array we get $P_i = i$ by the derangement. This can be easily implemented using the next_permutaion and single for loop in C++.
The only case that remains is when $N = 3$.The only edge case in which an answer exists and it is

not possible to get the answer using the approach defined above is $P = [3, 2, 1]$. This case can be solved beforehand and taken into account.

Answer for P={3,2,1}

2
1 2
2 3 1
1 3
1 2 3

TIME COMPLEXITY:

$O(N)$ or $O(N^2)$ depending on implementation for each test case.

SOLUTION:

Setter's solution
Editorialist's solution

Quote

shubu9211 **Jun '22**

An alternative approach to the problem
: **https://www.codechef.com/START41A/problems/DEARRANGE**

Read the editorial's solution for the test cases in which the answer will be 0 or 1 and also one edge test case.

Now, for case when $ans = 2$:

Rotate the array by $1, n$ times and check whether the current rotated array is valid or not each time. (Valid array: If it does not match with any index of the given array and also with the sorted array).

Now, there may be a possibility that none of the rotated arrays is valid. Think of the case when will it be possible 🙂

Since after each rotation, so there must be at least one index in each rotation that matches with the given array or with the sorted array, so all the indices follow some fixed order (i.e. after x rotations yth indexed element will make the array invalid).

How can we disturb this order?

Yes, it's easy, swap any two elements to disturb the order → Just find two elements that do not match with their index, and also they should not match with their index after the swap.

The order is disturbed, now we can surely say that at least in one of the n rotations we will get the valid array.

In $2nd$ step, just print the sorted array.

Submission : **https://ideone.com/4S9T4y**

---

hackapie **Jun '22**

```
if(a[ids[i]]==j || a[ids[j]]==i)
 {
     swap(a[ids[i]],a[ids[j]]),done=1;
     break;
}
```

Can you explain this if condition for swapping in your code?

---

hackapie **Jun '22**

Also i tried to implement your logic by doing this:
→ if we have an array where each rotation is invalid then this implies in each rotation some element is going to its required position in sorted array. And these elements are going to be distinct for each rotation. This implies we can calculate distances for each element with respect to its position in current permutation and sorted array and this distance array must be a permutation of [0,n-1]
→ now if this distance array is the permutation of [0,n-1] then we need to swap and update distance

array
→ now for rotation we want to make sure that no element comes back to its sorted position so we need to make some rotation which is not present in the distance array. Also if we did the swapping above then if we swapped (i,j) we can’t rotate by j-i since it won’t conflict with sorted array but it will with initial permutation. So we need to find such rotation value and rotate

Do you think there is a mistake here?

---

wuhudsm **Jul '22**

Why the method using bipartite graph matching algorithm gets WA?
https://www.codechef.com/viewsolution/65912677

PREREQUISITES:

Frequency arrays

PROBLEM:

Given an array A, find the minimum number of elements that need to be deleted from it so that the pairwise xor of any two elements in the resulting array is ≤ 1.

EXPLANATION:

For each $0 \leq x \leq N$, let $freq(x)$ denote the frequency of x in A, i.e, the number of times it occurs. We'll use this later.

First, let's find out what the final array can look like.
Suppose it contains two elements x and y. Then, the condition tells us that either $x \oplus y = 0$ or $x \oplus y = 1$.

- If $x \oplus y = 0$, then the only possibility is that $y = x$.
- If $x \oplus y = 1$, then the only possibility is that $y = x \oplus 1$.

So, if the final array contains x, the only elements it can contain are x and $x \oplus 1$.

Suppose we fix x. Then, our only option is to delete every element that is not x or $x \oplus 1$.
Recall the frequency array we initially constructed: it tells us that the number of such elements is exactly $N - freq(x) - freq(x \oplus 1)$.

The final answer is hence simply the minimum value of $(N - freq(x) - freq(x \oplus 1))$ across all $0 \leq x \leq N$, which can be easily found by just iterating across x once $freq$ has been precomputed.

TIME COMPLEXITY

$O(N)$ per test case.

PREREQUISITES:

Strings

PROBLEM:

A bracket sequence S is called *dense* if one of the following is true:

- S is empty.
- $S = (X)$ where X is dense.

You are given a bracket sequence S. What is the minimum number of brackets you must remove to make it dense?

EXPLANATION:

In order to solve this problem we can keep a count of all the opening brackets and closing brackets for each position in the string. First we will calculate the number of closing brackets in the string. This would be denoted by $close$

The maximum answer can be n as we would get a dense string if we remove all the brackets. Now we traverse from 1 to n. Initially we would have 0 opening brackets(denoted by $open$) and $close$ closing brackets. At each position there can be two cases:

- $s[i] = ($: Here we increment $open$ by 1. Thus we have $open$ opening brackets in the left and $close$ closing brackets at the right, so the maximum length of dense string that can be formed at this position is $min(2 \times open, 2 \times close)$. Thus,

$$ans = min(ans, n - min(2 \times open, 2 \times close))$$

- $s[i] =)$: Here we simply decrement $close$ by 1.

By the end of our traversal we will have our answer.

TIME COMPLEXITY:

$O(N)$, for each test case.

PROBLEM:

Given a binary string S, in one move you can insert either 0 or 1 at any position. Find the minimum number of moves so that the resulting string has no adjacent equal characters.

EXPLANATION:

Note that a single move allows us to 'break apart' exactly one pair of equal adjacent characters, by inserting either 1 between 00 or 0 between 11.
Further, this move doesn't create any new equal adjacencies.

So, the answer is simply the number of pairs that are already adjacent and equal, i.e, positions i $(1 \leq i < N)$ such that $S_i = S_{i+1}$, which can be computed with a simple `for` loop.

TIME COMPLEXITY

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

PREREQUISITES:

Greedy algorithms, observation

PROBLEM:

There are N events and M participants competing in them. The j-th participant scored $A_{i,j}$ in the i-th event.

For a fixed subset S of events, its *differentiation* is defined as follows:

- First, the score of the j-th participant is $s_j = \sum_{x \in S} A_{x,j}$
- Then, the differentiation is $\max_{i=1}^{M}(\sum_{j=1}^{M} |s_i - s_j|)$

Find the maximum possible differentiation across all possible subsets of events.

EXPLANATION:

First, let us analyze the formula for differentiation. Say a set S of events has been fixed, and we compute the scores $s_1, s_2, \ldots, s_M$. Without loss of generality, $s_1 \leq s_2 \leq \ldots \leq s_M$.

What is the differentiation in this case?

Answer

It is either $(s_1 - s_1) + (s_2 - s_1) + \ldots + (s_M - s_1)$ or $(s_M - s_1) + (s_M - s_2) + \ldots + (s_M - s_M)$. That is, if $X = s_1 + s_2 + \ldots + s_M$, then the differentiation is the larger of $X - M \cdot s_1$ and $M \cdot s_M - X$.

This tells us the following: given any subset, its differentiation is decided by either its largest score or its smallest score.

Now we have something else to work with. Instead of fixing the subset of events chosen, let us fix the person with minimum score, and calculate the maximum possible differentiation that can be achieved with this person as the minimum. Similarly, we can compute the maximum possible differentiation assuming the person has the highest score. Knowing these two values for everyone, the final answer is simply the maximum of all of them.

So, fix a person j. We want to compute the maximum possible differentiation assuming this person has the minimum score (the case where they have the maximum score can be computed similarly). This can be done as follows:

How?

Recall that the formula for differentiation when s_j is the minimum, is $\sum_{i=1}^{M} s_i - M \cdot s_j$.

Each s_i is simply the sum of scores of person i in all chosen events. So, rewriting the above sum in terms of the set S of chosen events, it can be seen to be

$$\sum_{x \in S} \sum_{i=1}^{M} A_{x,i} - M \cdot \sum_{x \in S} A_{x,j} = \sum_{x \in S} \sum_{i=1}^{M} (A_{x,i} - M \cdot A_{x,j})$$

So, each event x contributes a value of $\sum_{i=1}^{M}(A_{x,i} - M \cdot A_{x,j})$ to the overall summation — note that this value depends only on x and j.

We want to maximize the sum, and as seen above the events can be chosen (or not chosen) independently. So, we simply compute the value of $\sum_{i=1}^{M}(A_{x,i} - M \cdot A_{x,j})$ for each event, and add it to the answer if it is positive.

Implementing the above solution as-is will result in a runtime of $O(N \cdot M^2)$. However, it can be optimized to $O(N \cdot M)$ by simply precomputing the value of $C_i = A_{i,1} + A_{i,2} + \ldots + A_{i,M}$, and then looking this up instead of constantly recomputing it.

Something to think about

You might notice that we started off the above solution by assuming that s_j was the smallest score. However, we never really enforced that condition anywhere afterwards, and simply took events as and when they contributed positively to the overall answer without actually checking if s_j was the minimum.

Does ignoring that condition during calculation actually affect the answer in any way?

TIME COMPLEXITY:

$O(N \cdot M)$ per test case.

PROBLEM:

Given N, construct a permutation of $\{1, 2, 3, \ldots, N\}$ such that no two adjacent prefixes have the same median.

Note that the median of a set of size $2K$ is its K-th element in sorted order.

EXPLANATION:

We can make the following observations:

- Suppose A is a set of **odd** size and median M. If we add another element, say x, to A:
  - If $x \geq M$, the median of $A \cup \{x\}$ still remains M
  - If $x < M$, the median of $A \cup \{x\}$ is not M
- Suppose A is a set of **even** size and median M. If we add x to A:
  - If $x \leq M$, the median of the new set is still M
  - If $x > M$, the median of the new set is not M

This tells us the following, if we have fixed the first i elements of the permutation:

- If i is odd, the $i+1$-th element must be smaller than the current median
- If i is even, the $i+1$-th element must be larger than the current median

One construction that satisfies the above properties is as follows:

Let $K = \lfloor \frac{N}{2} \rfloor$. Then,

- Place $K+1, K+2, \ldots, N$ in the odd positions
- Place $K, K-1, \ldots, 1$ in the even positions

For example, if $N = 11$ (and so $K = 5$), the permutation looks like $[6, 5, 7, 4, 8, 3, 9, 2, 10, 1, 11]$.

TIME COMPLEXITY:

$O(N)$ per test case.

PREREQUISITES:

Shortest-path algorithms (either Dijkstra or Floyd-Warshall)

PROBLEM:

You have an $N \times N$ matrix A, whose *disgust* is defined to be $\sum_{i=1}^{N} \sum_{j=1}^{N} (A_{i,j} - A_{j,i})^2$.
You also have M changes, the i-th of which allows you to replace a single occurrence of x_i with y_i for some cost c_i.

Find the minimum possible sum of cost + disgust for A.

EXPLANATION:

Note that each pair of 'opposite' cells of A contribute to the answer independently. So, it is enough to solve the problem for a single pair of opposite cells, then apply this solution to every pair.

In other words, the problem can be restated as follows: you have 2 integers x_1 and y_1. You can apply some of the given operations on them to convert them to x_2 and y_2 respectively. Find the minimum possible value of cost + $2 \cdot (x_2 - y_2)^2$.

There are several different ways to do this, a few of which are detailed below.

Dijkstra

Let $f(x, y)$ denote the minimum value assuming we start with x and y. If we are able to precompute this value for every pair of (x, y), the problem is obviously solved.

Note that $f(x, y)$ can be defined as follows:

- If no changes are made, the value is $2 \cdot (x - y)^2$
- Otherwise, either x or y is changed.
  - Suppose x is changed to z with cost c. Then, we get a value of $c + f(z, y)$
  - Suppose y is changed to z with cost c. Then, we get a value of $c + f(x, z)$
- Clearly, finding the minimum value of this across all z will give us the answer.

This formulation is suspiciously like a shortest path problem: we can think of $f(x, y)$ as the 'shortest path' to reach (x, y), and changing either x or y equates to following an edge.
Also, the initial distance to vertex (x, y) is $2 \cdot (x - y)^2$ and not infinity.

A problem like this can be solved with the help of multisource dijkstra. For each $1 \leq x, y \leq N$, set its distance to $2 \cdot (x - y)^2$ and insert it into a priority queue/set. Now, run the standard dijkstra algorithm, and the end result is the correctly computed values of $f(x, y)$.
Notice that the edges in this case are reversed compared to what is given in the input, i.e, if the input allows you to convert i to j with a cost of c, then you create the edge $j \rightarrow i$ with weight c.

There are N^2 vertices and $\mathrm{O}(N)$ edges corresponding to each vertex, for a total of $\mathrm{O}(N^3)$ edges. So, this runs in $\mathrm{O}(N^3 \log N)$ time.

The tester's code below implements this.

Floyd-Warshall

Suppose we start with (x_1, y_1) and end at (x_2, y_2). There are only N different values of x_2, so let's try each of them and try to find the optimal y_2 once x_2 is fixed.

The cost can be computed as follows:

- First, the cost to change x_1 to x_2, say this is $c(x_1, x_2)$.
- Second, the cost to change y_1 to y_2, say $c(y_1, y_2)$
- Finally, the value $2 \cdot (x_2 - y_2)^2$

Computing all the values of $c(x, y)$ is not too hard: $c(x, y)$ is exactly the shortest path from x to y in the graph defined by the input edges. Computing this for every (x, y) pair can be done in $\mathrm{O}(N^3)$ using Floyd-Warshall.

Now, the above cost can be written as $c(x_1, x_2) + 2x_2^2 + (c(y_1, y_2) - 4x_2y_2 + 2y_2^2)$. The first two terms are constants once x_1 and x_2 is fixed, and the third term is of the form $kx_2 + m$, where k and m are constants that depend on y_1 and y_2.

Minimizing such an expression can be done with the help of the **convex hull trick** by constructing N such containers and querying the one corresponding to y_1.
This allows us to compute the answer in $\mathrm{O}(N \log N)$ for a given (x_1, y_1) pair, which is fast enough to solve the problem since there are only $\mathrm{O}(N^2)$ such pairs. The editorialist's code implements this approach.

However, there is also a solution that doesn't need any fancy data structures.

Consider another function $d(x, y)$, defined as follows:

- Suppose one of our numbers is fixed to be x, and we now want to change y. What is the minimum cost of doing this?

By considering every possible value that y can be turned into, It is easy to see that

$$d(x, y) = \min_{z=1}^{N}(c(y, z) + 2 \cdot (x - z)^2)$$

Once $c(x, y)$ has been computed, $d(x, y)$ can also be trivially computed in $\mathrm{O}(N^3)$.

Now, to answer the query for (x_1, y_1):

- Fix the value x_2 that x_1 will be turned into. This has cost $c(x_1, x_2)$.
- Now that x_2 is fixed, we need to convert y_1 to something else. By definition, the minimum cost here is exactly $d(x_2, y_1)$.
- So, the answer is the minimum of $c(x_1, x_2) + d(x_2, y_1)$ across all $1 \leq x_2 \leq N$.

This gives us a solution in $\mathrm{O}(N^3)$.

TIME COMPLEXITY

$\mathrm{O}(N^3)$ per test case.

PROBLEM:

You have a binary string S of length N. Find the maximum possible xor of two of its substrings of equal length. The substrings **cannot** overlap.

EXPLANATION:

Once again, we take care of some obvious corner cases first: if S contains only 0's or only 1's, the answer is 0.

Now, to maximize the answer, as always we try to first maximize its length.

Finding the length of the answer can be done easily in $\mathrm{O}(N^2)$, as follows:

- Iterate across all pairs (i,j) such that $1 \leq i < j \leq N$ and $S_i \neq S_j$.
- Assume that the first substring starts at i and the second at j.
- The maximum length we can get from these two is then $\min(j-i, N-j+1)$.
- The length of the answer is the maximum of this value across all valid pairs (i,j).

This tells us the length of the answer, say K. Now we need to look at all pairs of disjoint substrings of length K. However, there can be $\mathrm{O}(N^2)$ such pairs, and finding the xor of each one would make us take $\mathrm{O}(N^3)$ time in total, which is of course too slow.

Instead, we can make some observations. Consider the left endpoints L_1, L_2 $(L_1 < L_2)$ of an optimal answer. Then,

- Either $L_2 = R_1 + 1$, i.e, $L_2 = L_1 + K$; or
- $R_2 = N$, i.e, $L_2 = N - K + 1$

Proof

Suppose neither of the above cases holds, i.e, $R_2 < N$ *and* $L_2 > R_1 + 1$.
Then, we can simply increase both R_1 and R_2 by 1 to get longer disjoint strings with a strictly higher xor (since the xor would at least multiply by 2), which is a contradiction.

So, one of the above two conditions *must* hold.

This brings down the number of candidate pairs to $\mathrm{O}(N)$ — once a left endpoint L_1 is fixed, there are only two potential candidates for L_2, so simply take both of them. Do this for each $1 \leq L_1 \leq N$.

Then, compute the xors of all $\leq 2N$ candidate pairs, and take the maximum across them all. This gives us a solution in $\mathrm{O}(N^2)$ (since xor computation and string comparison are both $\mathrm{O}(N)$).

Once again, note that the output is the decimal value of the answer, modulo $10^9 + 7$. This can be computed in $\mathrm{O}(N)$, and the method to do so is detailed in the editorial for **JOINTXOR**.

TIME COMPLEXITY

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

PROBLEM:

Given an array A containing N distinct integers ranging from 1 to $2N$, perform the following operation exactly K times:

- Choose some $1 \leq x \leq 2N$ such that $x \notin A$, append x to A, and add $\max(A) - x$ to your score

What is the maximum possible final score?

EXPLANATION:

Let's look at how the score changes when we add a new element x to A. There are two cases:

- If $x < \max(A)$, then we get $\max(A) - x$ added to the score
- Otherwise, we must have $x = \max(A)$, in which case we add 0 to the score.

Ideally, we'd like as many operations of the first type as possible, since they're profitable. Note that the score of the first operation is maximized when x is as small as possible and $\max(A)$ is as large as possible.

This gives us a strategy. Let M denote the initial maximum of A, before any operations are done.

- Suppose we add K elements that do not change the maximum, i.e, they are all $< M$. Then, of course we choose the smallest K elements to insert.
- Suppose we add some element that does change the maximum.
  - It's optimal to add this element first so that all later operations have a higher score.
  - It's optimal to add as large a maximum as possible, so let's just choose $2N$ to add in the first move
  - The other $K - 1$ elements then should be chosen to be as small as possible to maximize the score, so choose the $K - 1$ smallest remaining elements

The answer is the maximum of both cases. Each one's score can be computed in $\mathrm{O}(N)$, giving us a linear time solution.

Note that we need to be able to quickly find the sum of the K smallest elements that are not in A. This can be done by maintaining a `mark` array that denotes which elements are already in A, then iterating across it from 1 to $2N$ and adding i to the sum if `mark[i] == 0`.

TIME COMPLEXITY

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

PROBLEM:

Given integers N and X, generate a **palindrome** of length N consisting of **lowercase English alphabets** such that the number of **distinct** characters in the palindrome is **exactly** X.
If it is impossible to do so, print -1 instead.

EXPLANATION:

Lets tackle this problem case by case:

- Case 1: When $N = 1$:
  Here the answer can be any character, example ‘a’.
- Case 2: When $N < (2 \times X) - 1$:
  Here it won’t be possible to construct the required string, since for X distinct characters we need to use at least $(2 \times X - 1)$ characters.
- Case 3: When $N = (2 \times X) - 1$:
  Assuming s to be a string of length X having all distinct characters, we can construct the following string as our answer:

$$s_1 s_2 \ldots s_{x-1} s_x s_{x-1} s_{x-2} \ldots . s_1$$

- Case 4: When $N > (2 \times X) - 1$:
  Assuming s to be a string of length X having all distinct characters, we can have three parts as:

$$\begin{gathered} left = s \\ mid = A\ string\ of\ length\ (N - 2 \times X)\ having\ all\ characters\ as\ s_1 \\ right = reverse(s) \end{gathered}$$

  Thus our final answer for this case would then be:

$$ans = left + mid + right$$

TIME COMPLEXITY:

$O(N)$, for each test case.

PROBLEM:

You are given an array A of length $2N$. Let B be an empty array. Repeat the following N times:

- Pick two elements x and y of A. Delete them from A and append $x-y$ to B.

Is it possible to perform moves so that $B_i \neq B_{i+1}$ for each $1 \leq i < N$?

EXPLANATION:

Suppose we perform moves on A in some order to end up with B. Let's analyze the array we get.

- If $B_i \neq B_{i+1}$ for any i, of course we're happy.
- Otherwise, $B_i = B_{i+1}$ for some index. There are two cases here:
  - If $B_i \neq 0$, we can actually fix this error. Note that $B_i = x-y$ for some x and y from A. Instead, we could've just chosen $B_i = y-x$ instead: now B_i and B_{i+1} have different signs, so they obviously can't be equal.
  - If $B_i = 0$, once again we have two cases:
    - Suppose $B_i = x-x$ and $B_{i+1} = y-y$, where $x \neq y$. Then, we can replace them with $x-y$ and $y-x$ instead, which fixes the problem.
    - However, if $B_i = B_{i+1} = x-x$, there's nothing we can do.

This should give us the intuition that if there are too many occurrences of the same integer x in A, we'll always end up with some adjacent pair in B that's both $x-x$, which can't be fixed.
But, how many is too many?

Let's fix an integer x, and say it occurs k times in A. Then,

- If $k \leq N$, there exists a way to create B such that we never create zeros of the form $x-x$, since we can always pair an x with a non-x.
- Otherwise, there will always be some pairs of the form $x-x$.
  - There are $2N-k$ non-x elements, allowing us to create at most $2N-k$ pairs of x with something else.
  - This leaves a minimum of $k-(2N-k) = 2k-2N$ instances of x, which means B will contain at least $k-N$ zeros of the form $x-x$.
  - For these zeros to not adjacent in B, their number cannot exceed half the length of B (which is N). That is, $k-N \leq \lceil \frac{N}{2} \rceil$ must hold ($\lceil\ \rceil$ denotes the **ceiling** function).

Now notice that if this condition holds for some k, it will also hold for any lower k. So, it's enough to check this for the largest value of k. That is,

- Let k be the largest number of occurrences of some element in A.
- Then, the answer is "Yes" if and only if $k-N \leq \lceil \frac{N}{2} \rceil$

TIME COMPLEXITY

$O(N)$ per test case.

PREREQUISITES:

Combinatorics, **Stars and bars**

PROBLEM:

Given N, x, y, find the number of arrays P satisfying the following:

- P contains exactly x ones, y twos, and $N - x - y$ threes
- $P_1 = P_N = 1$
- No two adjacent elements of P are equal.

EXPLANATION:

This is a pretty much a purely combinatorial task.

The main observation is as follows:
Consider two adjacent ones in any valid array P. Then, the subarray between them must consist of alternating twos and threes.
In particular, if we fix the first element of this subarray, that fixes the entire subarray.

Now, note that any subarray between two ones will be in one of four forms:

- Even length, and starting with either 2 or 3
- Odd length, and starting with either 2 or 3

The even length arrays are functionally equivalent since they contain an equal number of twos and threes. However, the odd subarrays aren't equivalent: the starting element appears one more time than the rest. Let's try to use this.

Suppose there are k_1 odd length subarrays that start with a 2, k_2 odd length subarrays that start with a 3, and k_3 even length subarrays. Then, we have the following equations:

- $k_1 + k_2 + k_3 = x - 1$, since every 1 except the last has one of these subarrays immediately following it
- $k_1 - k_2 = y - z$, since each odd-length subarray starting with 2 contributes one extra 2, and each odd-length subarray starting with 3 contributes one extra 3.

Let's fix the value of k_1 (it can be anything from 0 to $x - 1$).
Note that the above equations allow us to compute uniquely the values of k_2 and k_3.
If either k_2 or k_3 are invalid (i.e, negative or $> x - 1$), then ignore this value of k_1.

So, suppose we know k_1, k_2, k_3. Let's try to count the number of arrays with these parameters.

That can be done in a few steps.

- First, there are $x - 1$ subarrays. Let's fix which ones of them are even length and which are odd: this gives us $\binom{x-1}{k_3}$ choices.
- Among the odd-length ones, let's fix which ones start with 2 and which ones start with 3: this gives us $\binom{k_1+k_2}{k_1}$ choices.
- The even-length subarrays can be arranged in two ways (starting with 2 or 3), so we have 2^{k_3} choices there.
- Finally, we need to count the number of ways to choose lengths for these subarrays. That is an application of the stars-and-bars technique.

How?

There are exactly $x - 1$ subarrays. Suppose the i-th of them has length a_i.

The total length of the subarrays must equal the number of twos and threes, i.e, y+z.
So,

$$a_1 + a_2 + \ldots + a_{x-1} = y + z$$

However, there are some more constraints:

- Exactly $k_1 + k_2$ of these values are odd. Note that we have already fixed which ones are odd with a binomial coefficient, so without loss of generality we can assume $a_1, a_2, \ldots, a_{k_1+k_2}$ are odd and the rest are even.
- The even values must be strictly positive (since if they had length 0, two ones would be adjacent).

Let's write the odd values as $a_i = 2b_i + 1$, and the even values as $a_i = 2 + 2b_i$. Plugging this into the first equation,

$$\begin{aligned}(2b_1 + 1) + \ldots + (2b_{k_1+k_2} + 1) + (2 + 2b_{k_1+k_2+1}) + \ldots + (2 + 2b_{x-1}) &= y + z \\ \implies 2b_1 + 2b_2 + \ldots + 2b_{x-1} &= y + z - (k_1 + k_2) - 2k_3 \\ \implies b_1 + b_2 + \ldots + b_{x-1} &= \frac{y + z - (k_1 + k_2) - 2k_3}{2}\end{aligned}$$

The b_i have no constraint, other than the fact that they have to be non-negative integers.
Any solution to this equation gives us a valid solution to the original equation by reversing the $a_i \to b_i$ process, so it's enough to count the number of solutions to this equation.
Do note that if $y + z - k_1 - k_2 - 2k_3$ is odd, then the equation has no solution.

Counting the number of solutions here is a direct application of stars and bars, and is simply $\binom{M+x-2}{M}$, where $M = \frac{y+z-(k_1+k_2)-2k_3}{2}$.

So, the final solution is to simply add up

$$\binom{x-1}{k_3}\binom{k_1+k_2}{k_1}\binom{M+x-2}{M}2^{k_3}$$

where $M = \frac{y+z-(k_1+k_2)-2k_3}{2}$, across all possible k_1.

Each binomial coefficient can be computed in $\mathrm{O}(1)$ if factorials and their inverses are precomputed, and the power of 2 can be computed in $\mathrm{O}(\log N)$ using binary exponentiation (or all required powers of 2 can be precomputed, since we only need powers $\leq N$ anyway).

This gives us a solution in $\mathrm{O}(N \log N)$.

TIME COMPLEXITY

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

PREREQUISITES:

Knowledge of GCD

PROBLEM:

You have an array A. In one move you can choose an index i and set $A_i \to \frac{A_i}{X}$ where X is some factor of A_i.
Find the minimum number of moves needed to make all the elements equal.

EXPLANATION:

Suppose we make some moves and end up with all the elements equal, say to an integer y.
Then, notice that y must divide *every* A_i, since one operation only allows us to replace an element with one of its factors.

Since y must divide every A_i, y must also divide $\gcd(A_1, A_2, \ldots, A_N)$.

Now notice that we might as well choose $y = \gcd(A_1, A_2, \ldots, A_N)$, because:

- It obviously satisfies the condition that it divides every element of A
- It is the largest possible integer that satisfies this condition

This gives a rather simple solution:

- Compute $y = \gcd(A_1, \ldots, A_N)$.
  - To do this, first initialize $y = A_1$. Then, for each $i > 1$, set $y = \gcd(y, A_i)$.
  - Computing the gcd of two elements can be done using inbuilt functions, like `std::gcd` in C++ and `math.gcd` in Python.
- Then, for each A_i:
  - If $A_i = y$, we don't need to change this element at all.
  - Otherwise, we need exactly one move to turn A_i into y.

So, count the number of times y appears in the array — let this be k. The answer is then $N - k$.

TIME COMPLEXITY

$O(N)$ per test case.

PREREQUISITES:

Factorization, Sieve of Eratosthenes, Dynamic programming, Binary search/2-pointers

PROBLEM:

Given an array A, in one move you can divide any element of it by one of its prime factors.
Find the minimum number of moves to make the array non-decreasing.

EXPLANATION:

A few greedy strategies might come to mind, but none of them really work.

Instead, let's try to use dynamic programming.
Let $dp_{i,x}$ denote the minimum number of moves to make $A_1 \leq A_2 \leq \ldots \leq A_i$, such that $A_i = x$.
Transitions are fairly easy to see:

$$dp_{i,x} = cost(A_i, x) + \min_{1 \leq y \leq x} dp_{i-1,y}$$

where $cost(a, b)$ denotes the smallest number of moves to convert a to b when dividing by primes.

There are a couple of things to do now:

- Figure out how to compute $cost(a, b)$
- Optimize the dp computation to work enough.

Computing $cost$

Note that $cost(a, b)$ depends solely on the prime factorizations of a and b. In particular,

- b must be a factor of a
- Exactly one prime can be removed from a at each step
- So, if $p(a)$ denotes the number of (not necessarily distinct) primes in the factorization of a, then $cost(a, b) = p(a) - p(b)$.
- Alternately, you can see that we remove exactly $\frac{a}{b}$ using primes, so this also equals $p(\frac{a}{b})$.

So, all that needs to be done is to compute $p(n)$ for every $1 \leq n \leq 5 \cdot 10^5$.
This is a standard computation that can be done with the help of a **sieve**.

Optimizing dp

There are two things to optimize: the number of states, and performing the transitions.

Note that $dp_{i,x}$ only matters for x that is a factor of A_i since no other x can be reached via division.
So, all other states can be ignored. This gives an easy upper bound of about $O(N\sqrt{M})$ states (where $M = 5 \cdot 10^5$), but can be in fact bounded by $200 \cdot N$, since no number in the given range has more than 200 factors.

This also makes our transitions take ≤ 200 steps each (since we only need to iterate over divisors of A_{i-1}, making the entire solution something like $O(N \cdot d^2)$ where $d = 200$.

Unfortunately, this is likely too slow, and we need to optimize it a bit. There are a couple ways to do this.

- One way is to use binary search along with prefix minimums. Note that $dp_{i,x}$ depends exactly on some prefix minimum of dp_{i-1}. So, if the prefix minimums of dp_{i-1} are computed, the appropriate one can be

found using binary search on the factors of A_{i-1}. This gives a solution in $\mathrm{O}(N \cdot d \log d)$.
- Instead of binary search, one can also use two pointers to achieve $\mathrm{O}(N \cdot d)$.

Finally, there is the issue of actually finding the factors. Once again, there are two ways to do this:

- Directly factorize each A_i in $\mathrm{O}(\sqrt{M})$.
- Or, once again use a sieve to precompute the divisors of all numbers $\leq M$.

The second method is faster and is what is intended, however the first method can also pass if written well. Note that you might TLE if you use both sqrt factorization *and* binary search, but optimizing any one of them should be enough to get AC.

TIME COMPLEXITY

$\mathrm{O}(M \log M)$ precomputation followed by $\mathrm{O}(N \cdot D)$ per test case where $M = 5 \cdot 10^5$ and $D \approx 200$ is the maximum number of factors a number $\leq M$ can have.

PREREQUISITES:

Frequency tables

PROBLEM:

Given an array A, report whether it has a dominant element, i.e, an element whose frequency is larger than the frequency of any other element.

EXPLANATION:

Let M be the maximum frequency of any element in A. Then,

- If there are two or more elements with frequency M, there cannot be a dominant element
- Otherwise, there is a unique element with frequency M, which is a dominant element

To check these cases, all that needs to be done is to know the frequency of every element of A. Note that computing frequencies by iterating across the entire array each time might be too slow, because it can be a complexity of $\mathrm{O}(N^2)$ in the worst case.

Instead, we build the *frequency array* of A. This can be done as follows:

- Let F be our frequency array. Since every element in A lies between 1 and N, it is enough to have F be of length N.
- Initialize F with zeroes.
- Then, for each i from 1 to N, add 1 to F_{A_i}.
- At the end of this, F_x holds the number of times x appears in A.

Now that we have F, we can solve the problem as mentioned in the first step. Let M be the maximum element of F, and then check whether M appears exactly once in F (in which case the answer is “Yes”) or more than once (in which case the answer is “No”).

TIME COMPLEXITY:

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

PROBLEM:

You are given a graph G representing connections in a company. You know that G has the following structure:

- For some integer M, there are M departments. Each department has a manager.
- All the managers are connected to each other
- All the members of a department are connected to each other
- No other connections exist

Find the number of departments, the managers of the departments, and which employees belong to which department.

In the easy version, you further know that each department has a minimum of two people in it.

EXPLANATION:

Easy version

Look at the vertex with the smallest degree, say u. If $M > 1$, u is guaranteed to not be a manager.

Why?

Consider the department u belongs to. Suppose it has k people.
The non-managers of this department will all have the same degree, i.e $k - 1$.
Note that if there are M departments, the manager will have degree $k - 1 + M - 1$, which is strictly more than $k - 1$ when $M > 1$.

Since every department has two people in it, there definitely exists a non-manager employee. So, u is definitely not a manager.

Note that when $M = 1$, it doesn't matter who is chosen as the manager. So, for our purposes we can always assume that u is not a manager (even though we don't know M yet).

Now, the above discussion tells us this:

- Suppose u has degree k. Then, $k - 1$ of its neighbors will also have degree k, being non-manager employees in the same department.
- Exactly one neighbor of u will have a degree higher than k. Let this be v. v is the manager of u's department.
- Once we know v, we know all the other managers: they are all the neighbors of v who are not neighbors of u. This also tells us the number of departments.
- Knowing all the managers allows us to form the departments in similar fashion: if w is a manager, the members of that department are then all the neighbors of w who are not managers.

This gives us all the information we need: we know the number of departments, the managers of each department, and the members of each department.

Hard version

The only difference here is that there might be some department with only one person. In particular, when looking at the vertex of lowest degree, they can possibly be a manager this time.

However, note that there are only two cases possible:

- First, if u is not a manager, the exact same solution as the easy version applies.
- Second, if u is a manager, then we know all the managers: they are simply all the neighbors of u. Just as above, knowing all the managers allows to reconstruct all the departments.

Try both cases!

Assume u is a manager, and try to reconstruct a solution. If you succeed, great! Print the solution you obtained and continue on.
If you fail, then the only possibility is that u is not a manager. Now apply the solution to the easy version and you are done.

Maximum degree

There is an alternate approach that looks at the maximum degree instead of the minimum degree, although it ends up being quite similar in the end.

Note that the vertex with maximum degree is always going to be a manager. So, some of its neighbors are all the other managers, and the remaining are the members of its department.

Notice that these two sets each form a clique, and are disjoint from each other. This allows us to compute these disjoint sets quite easily. Then, check for each one of them whether there is a valid solution with that set as the managers.

TIME COMPLEXITY

$O(N^2)$ per test case.

PROBLEM:

You are given a graph G representing connections in a company. You know that G has the following structure:

- For some integer M, there are M departments. Each department has a manager.
- All the managers are connected to each other
- All the members of a department are connected to each other
- No other connections exist

Find the number of departments, the managers of the departments, and which employees belong to which department.

In the easy version, you further know that each department has a minimum of two people in it.

EXPLANATION:

Easy version

Look at the vertex with the smallest degree, say u. If $M > 1$, u is guaranteed to not be a manager.

Why?

Consider the department u belongs to. Suppose it has k people.
The non-managers of this department will all have the same degree, i.e $k - 1$.
Note that if there are M departments, the manager will have degree $k - 1 + M - 1$, which is strictly more than $k - 1$ when $M > 1$.

Since every department has two people in it, there definitely exists a non-manager employee. So, u is definitely not a manager.

Note that when $M = 1$, it doesn't matter who is chosen as the manager. So, for our purposes we can always assume that u is not a manager (even though we don't know M yet).

Now, the above discussion tells us this:

- Suppose u has degree k. Then, $k - 1$ of its neighbors will also have degree k, being non-manager employees in the same department.
- Exactly one neighbor of u will have a degree higher than k. Let this be v. v is the manager of u's department.
- Once we know v, we know all the other managers: they are all the neighbors of v who are not neighbors of u. This also tells us the number of departments.
- Knowing all the managers allows us to form the departments in similar fashion: if w is a manager, the members of that department are then all the neighbors of w who are not managers.

This gives us all the information we need: we know the number of departments, the managers of each department, and the members of each department.

Hard version

The only difference here is that there might be some department with only one person. In particular, when looking at the vertex of lowest degree, they can possibly be a manager this time.

However, note that there are only two cases possible:

- First, if u is not a manager, the exact same solution as the easy version applies.
- Second, if u is a manager, then we know all the managers: they are simply all the neighbors of u. Just as above, knowing all the managers allows to reconstruct all the departments.

Try both cases!

Assume u is a manager, and try to reconstruct a solution. If you succeed, great! Print the solution you obtained and continue on.
If you fail, then the only possibility is that u is not a manager. Now apply the solution to the easy version and you are done.

Maximum degree

There is an alternate approach that looks at the maximum degree instead of the minimum degree, although it ends up being quite similar in the end.

Note that the vertex with maximum degree is always going to be a manager. So, some of its neighbors are all the other managers, and the remaining are the members of its department.

Notice that these two sets each form a clique, and are disjoint from each other. This allows us to compute these disjoint sets quite easily. Then, check for each one of them whether there is a valid solution with that set as the managers.

TIME COMPLEXITY

$O(N^2)$ per test case.

PREREQUISITES:

Factorisation, Prefix Sums, Binary Search

PROBLEM:

You are given a list of N integers $A_1, A_2, \cdots, A_n$, along with Q queries.

Each query consists of two integers p and k.

Suppose you are allowed to reorder only those elements of A which are divisible by p.

You have to output the maximum possible value of $\sum_{i=1}^{k} A_i$ over all such reorderings.

EXPLANATION:

Our reordering strategy is trivial.

Let $S = \{A_i | \ i \in [N], A_i \text{ is divisible by } p\}$. We would like to place the largest element from S at the smallest index, the second largest element at the second smallest index, so on and so forth. \

What would be the value of $\sum_{i=1}^{k} A_i$ in such a reordering?

Define $T = \{i \in [k] \mid A_i \text{ is divisible by } p\}$.

Then, in our reordering, we would place the largest $|T|$ elements from S into indices in T, and we would place the elements that are already in those T indices somewhere else (possibly outside the range $[1, k]$).

Therefore, the answer we get would be $\sum_{i=1}^{k} A_i + (\text{sum of the largest } |T| \text{ elements of } S) - \sum_{i \in T} A_i$

We can precompute the factorisation of all numbers in range $[1, 1e5]$, and use this to prime factorise M. Note that each element of M can have at most $\log 1e5 < 17$ prime factors. Now, for each prime number $\in [1, 1e5]$, we can precompute S. We can precompute suffix sums of S (when sorted by value) and prefix sums of S (when sorted by index). Finally, we can precompute prefix sums of M.

Now we describe how to answer each query. We can find $|T|$ by doing a simply binary search for k in S, this takes $O(\log n)$. We can compute $\sum_{i=1}^{k} A_i$ in $O(1)$ using the prefix sums we precomputed. We can compute $(\text{sum of the largest } |T| \text{ elements of } S)$ in $O(1)$ using the suffix sums we precomputed. And finally, we can compute $\sum_{i \in T} A_i$ using the prefix sums of S in $O(1)$, that we also precomputed.

TIME COMPLEXITY:

Approximately $O(M \log M + 1e5 \log 1e5)$ precomputation, and then $O(\log N)$ per query, where $M = N \log 1e5$.

PROBLEM:

Chef considers a permutation P of $\{1, 2, 3, \ldots, N\}$ `End Sorted` if and only if $P_1 = 1$ and $P_N = N$.

Chef is given a permutation P.

In one operation Chef can choose any index i $(1 \leq i \leq N - 1)$ and swap P_i and P_{i+1}. Determine the minimum number of operations required by Chef to make the permutation P `End Sorted`.

Note: An array P is said to be a permutation of $\{1, 2, 3, \ldots, N\}$ if P contains each element of $\{1, 2, 3, \ldots, N\}$ exactly once.

EXPLANATION:

Let id_1 be the position of 1 in the permutation and id_N be the position of N in the permutation. We want 1 at position 1 and N at position N in the permutation. If id_1 is smaller than id_N we will have to make at least $id_1 - 1$ swaps to get 1 at position 1 and $N - id_N$ swaps to get N at position N in the permutation. But if $id_1 > id_n$ we will have to do $id_1 - 1$ swaps to get 1 at position 1 in the permutation. One of these swaps, swaps N and 1 hence id_N is increased by 1. Therefore the answer in this case is $id_1 - 1 + N - (id_N + 1)$.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Depth first search, Segment/fenwick trees, Binary search

PROBLEM:

You have a tree T, each node of which may or may not be activated.
Each node also has an energy, initially 0.
At time i, each activated node gives i energy to itself and its neighbors.

Answer Q queries of the form (u, T, K), with the answer being the minimum time after which at least T nodes on the path $1 \to u$ have an energy of $\geq K$.

EXPLANATION

This editorial will be for the hard version of the problem, with $K \leq 10^{18}$.

Let's focus on a single query (u, T, K). How to solve it?

To do that, let's look at how the energy of a vertex changes.
For vertex x, let m_x be the sum of A_v across all vertices v such that $v = x$ or v is a neighbor of x.
Then, vertex x will receive exactly $m_x \cdot i$ energy at time i.
In particular, at time i, vertex x will have $m_x \cdot (1 + 2 + \ldots + i) = m_x \cdot i \cdot (i + 1)/2$ energy.

So, suppose we know the m_x values for all vertices. To answer a query (u, T, K):

- First, note that the number of vertices with energy $\geq K$ is a non-decreasing function of time, so we can binary search on it to find the first time this number exceeds T.
  Be careful when choosing the upper limit of the binary search
- This requires us to be able to quickly calculate the number of nodes on the $1 \to u$ path that have $\geq K$ energy at a given time t.
- This means we want the count of nodes x such that $m_x \cdot t \cdot (t + 1)/2 \geq K$.
  t is a constant so by bringing this to the other side we get an equation of the form $m_x \geq c$ for some constant c.
- Keeping all the values of m_x from the root to u in an appropriate data structure (for example, a segment tree) allows us to find this count in $\mathrm{O}(\log N)$.
- So, a single query can be solved in $\mathrm{O}(\log N \log M)$ where $M = 10^9$.

This solves a single query. Now we extend this to multiple queries.

Note that a query at u only depends on the m_x values on the $1 \to u$ path. So, we can do the following and solve queries offline:

- Group queries by their u.
- Run a dfs on the tree.
- When you enter a vertex u, insert m_u into the data structure you are using.
- Then, solve all queries involving u.
- Next, continue on with the dfs to u's children.
- Finally, when exiting u, remove m_u from the data structure.

You can see that this process ensures that whenever answering a query at u, the data structure contains only those m_x values for x on the $1 \to u$ path, which is exactly what we want.

It is also possible to solve the queries online (for example, with a persistent segment tree).

TIME COMPLEXITY

$O(N + Q \log N \log M)$ per test case.

PREREQUISITES:

Depth first search, Segment/fenwick trees, Binary search

PROBLEM:

You have a tree T, each node of which may or may not be activated.
Each node also has an energy, initially 0.
At time i, each activated node gives i energy to itself and its neighbors.

Answer Q queries of the form (u, T, K), with the answer being the minimum time after which at least T nodes on the path $1 \to u$ have an energy of $\geq K$.

EXPLANATION

This editorial will be for the hard version of the problem, with $K \leq 10^{18}$.

Let's focus on a single query (u, T, K). How to solve it?

To do that, let's look at how the energy of a vertex changes.
For vertex x, let m_x be the sum of A_v across all vertices v such that $v = x$ or v is a neighbor of x.
Then, vertex x will receive exactly $m_x \cdot i$ energy at time i.
In particular, at time i, vertex x will have $m_x \cdot (1 + 2 + \ldots + i) = m_x \cdot i \cdot (i + 1)/2$ energy.

So, suppose we know the m_x values for all vertices. To answer a query (u, T, K):

- First, note that the number of vertices with energy $\geq K$ is a non-decreasing function of time, so we can binary search on it to find the first time this number exceeds T.
  Be careful when choosing the upper limit of the binary search
- This requires us to be able to quickly calculate the number of nodes on the $1 \to u$ path that have $\geq K$ energy at a given time t.
- This means we want the count of nodes x such that $m_x \cdot t \cdot (t + 1)/2 \geq K$.
  t is a constant so by bringing this to the other side we get an equation of the form $m_x \geq c$ for some constant c.
- Keeping all the values of m_x from the root to u in an appropriate data structure (for example, a segment tree) allows us to find this count in $\mathrm{O}(\log N)$.
- So, a single query can be solved in $\mathrm{O}(\log N \log M)$ where $M = 10^9$.

This solves a single query. Now we extend this to multiple queries.

Note that a query at u only depends on the m_x values on the $1 \to u$ path. So, we can do the following and solve queries offline:

- Group queries by their u.
- Run a dfs on the tree.
- When you enter a vertex u, insert m_u into the data structure you are using.
- Then, solve all queries involving u.
- Next, continue on with the dfs to u's children.
- Finally, when exiting u, remove m_u from the data structure.

You can see that this process ensures that whenever answering a query at u, the data structure contains only those m_x values for x on the $1 \to u$ path, which is exactly what we want.

It is also possible to solve the queries online (for example, with a persistent segment tree).

TIME COMPLEXITY

$O(N + Q \log N \log M)$ per test case.

PROBLEM:

Given an array A of length N, decide whether it can be partitioned into two arrays B and C that have an equal number of distinct elements.

EXPLANATION:

The answer is “No” if and only if N is odd *and* all the elements of A are distinct.

Proof

Let N be odd and all the elements of A be distinct. Then, no matter how the partition is done, B and C will have different sizes (and their number of distinct elements equals their size, so equality is impossible).

Now we have to prove that a suitable division always exists in the other case.

- Suppose A has an even number of distinct elements — for convenience, let’s call them $1, 2, 3, \ldots, 2K$. Then, put all occurrences of $1, 2, \ldots, K$ into B and all occurrences of $K+1, K+2, \ldots, 2K$ into C. B and C now have K distinct elements each, as required.
- Suppose A has an odd number of distinct elements — let them be $1, 2, \ldots, 2K+1$. Note that, in particular, $2K+1 < N$, so some element occurs greater than once — let one such element be x. Now, create B and C as in the even case by assigning all elements other than x. Finally, place one copy of x in B and the others in C. B and C both have $K+1$ distinct elements now.

This completes the proof.

Checking the above condition can easily be done with a frequency table in $\mathrm{O}(N)$ since the array elements are $\leq N$.

TIME COMPLEXITY

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

PROBLEM:

Given two binary strings A and B, in one move you can either flip one character of A, or reverse A.
Using at most $\lceil \frac{N}{2} \rceil$ moves, make A and B have the same number of inversions.

EXPLANATION:

There can be several strings with the same number of inversions as B, so we need to decide which one of them to convert to.

One simple case is B itself: if we can convert A to B within $\lceil \frac{N}{2} \rceil$ moves, that obviously gives us a solution. One trivial way to do this is to simply flip all those positions that differ in A and B.

Unfortunately, this is not always possible: a simple example is when $A = 0000 \ldots 000$ and $B = 1111 \ldots 1111$, when you need N moves no matter what.

However, this gives us a starting point. Suppose we can't convert A to B using our trivial algorithm. Then, we know for sure that A and B differ at $> \lceil \frac{N}{2} \rceil$ positions — in other words, they match at $< \lceil \frac{N}{2} \rceil$ positions. Let's try to use this to our advantage.

For a binary string S, let $rev(S)$ denote the reverse of S, and $flip(S)$ denote the string where we flip *every* character of S. What can you say about the number of inversions of $rev(S)$ and $flip(S)$?

Answer

They are equal!

A simple proof is to look at some pair of positions (i, j) with $i < j$.

- If $S_i = S_j$, then (i, j) doesn't contribute to an inversion in $flip(S)$ and $(N + 1 - j, N + 1 - i)$ isn't an inversion in $rev(S)$
- If $S_i < S_j$, then (i, j) is an inversion in $flip(S)$ and $(N + 1 - j, N + 1 - i)$ is an inversion in $rev(S)$
- If $S_i > S_j$, then (i, j) is not in inversion in $flip(S)$ and $(N + 1 - j, N + 1 - i)$ is not an inversion in $rev(S)$

This gives us a bijection between inversions in $flip(S)$ and $rev(S)$, so they are equal in number.

Now, let's go back to our initial solution. Since A and B match at $< \lceil \frac{N}{2} \rceil$ positions, A and $flip(B)$ differ at $< \lceil \frac{N}{2} \rceil$ positions, which means we can easily convert A to $flip(B)$ and have at least one move remaining.

Then, from the discussion above, note that $rev(flip(B))$ has the same number of inversions as $rev(rev(B))$. $rev(rev(B))$ is, however, just B.

So, using our one remaining move to reverse the string converts A to $rev(flip(B))$, which has the same number of inversions as B, and we are done.

TIME COMPLEXITY

$O(N)$ per test case.

PROBLEM:

You have an array of N integers and an integer K. In one move, you can choose any subarray and make every element of this subarray equal to the middle element (the left one, if length is even).

Can you make every element equal K?

EXPLANATION:

If some value is not in the array, there's no way to bring it into the array. So, if A doesn't contain K, the answer is immediately `No`.

So, suppose A contains K. In particular, suppose $A_i = K$. Then,

- If $i + 1 < N$, then choosing the subarray $[i, i + 1]$ allows us to set $A_{i+1} = K$.
- If $1 < i < N - 1$, then choosing the subarray $[i - 1, i + 1]$ allows us to set $A_{i-1} = A_{i+1} = K$.

In particular, if we have a K at some position i:

- Every position after i can be made K.
- Every position before i can be made K as long as $i + 1 < N$.

So, if there exists a position $i < N$ such that $A_i = K$, then the answer is immediately `Yes`.

That leaves us with the case when $A_N = K$, and this is the only K in the array. It's not hard to see that in this case, no other element can ever be made K.

Thus, the answer in this case is `Yes` if $N = 1$ and `No` otherwise.

TIME COMPLEXITY

$O(N)$ per test case.

PREREQUISITES

Dynamic Programming, Segment Tree

PROBLEM

You have an array A containing all the natural numbers from 1 to N twice. Find the number of **ordered pairs** of **disjoint** subsequences (P, Q) of A such that they are equal.

EXPLANATION

Let,

- dp_i : − denote the number of **ordered pairs** of subsequences (P, Q) that can be formed taking elements from index 0 to i.
- f_i, s_i : − denote the indices of the first and the second occurrence of A_i.
- t_i : − denote the number of **unordered pairs** of subsequences (P, Q) ending at indices f_i and s_i respectively.

Now, let's say we have already computed dp_{i-1}.

Case 1: If $i = f_i$,

No additional pair of (P, Q) can be formed by including the element at index i. So we can simply say, $dp_i = dp_{i-1}$.

Case 2: If $i = s_i$,

We have to add all the ordered pairs of subsequences (P, Q) ending at f_i and i to dp_{i-1}. So we can say,

$$dp_i = dp_{i-1} + 2 \cdot t_i.$$

Let's try to find t_i now. This comprises of 3 parts:

- **Part 1:** We can add A_{f_i} and A_i respectively at the end of all the **ordered pairs** of subsequences (P, Q) formed from indices 0 to $f_i - 1$.
- **Part 2:** We can add A_{f_i} and A_i at the end of all the **unordered pairs** of subsequences (P, Q) which end at f_x and s_x repectively, such that, $0 \leq f_x < f_i < s_x < i$.
- **Part 3:** An additional subsequence pair $([A_{f_i}], [A_{s_i}])$.

The **first part** is simply dp_{f_i-1} and as f_i is the first occurrence of A_i, we can also say $dp_{f_i-1} = dp_{f_i}$ *(Case 1)*.

Now, for the **second part**, you have f_i and i as fixed and you need to sum all the values of t_x where, $f_x < f_i < s_x < i$. So, let's say we have maintained an array T (say) till index $i - 1$ with the value t_x at the index of the first occurrence and $-t_x$ at the index of the second occurrence of all the elements having $s_x \leq i - 1$.

The sum of this array T from $j = 0$ to $j = f_i - 1$ will give us the value of the second part because in case of $s_x < f_i$, the values t_i and $-t_i$ are simply going to cancel each other out.

For the condition of $s_x < i$ to be fulfilled at all times, we will update the array T only as we iterate over i.
We **cannot** pre-compute the values of T, otherwise, the summation will also include values where $f_x < f_i$ but

$s_x > i$.

Mathematically,

$$T \begin{cases} T_{f_x} = T_{s_x} = 0, & \text{if } i - 1 < s_x \\ T_{f_x} = t_x \text{ and } T_{s_x} = -t_x, & \text{if } i - 1 >= s_x \end{cases}$$

and, $t_i = dp_{f_i} + \sum_{j=0}^{j=f_i-1} T_j + 1$.

Finally, we can update the array T as follows:
$T_{f_i} = t_i$ and $T_i = -t_i$

These range sum and point update queries in T can be handled efficiently using a Segment Tree.

Note: No updates in T were required in **Case 1** as $i < s_i$.

TIME COMPLEXITY

The time complexity is $O(N \cdot \log(N))$.

PROBLEM:

Chef calls a pair of integers (A, B) *equivalent* if there exist some **positive** integers X and Y such that $A^X = B^Y$.

Given A and B, determine whether the pair is *equivalent* or not.

EXPLANATION:

In order for two number A and B to satisfy the relation

$$A^x = B^y$$

where x and y are two positive integers, it needs to satisfy the following conditions:
1.) Both the number should have the same prime factors.
2.) The ratio of frequency of each prime factors of the two numbers should be the same.

Thus we would first check if both the numbers have same prime factors and if yes then we would calculate the frequency of each prime factor for the two numbers and then check if each pair of frequencies have the same ratio.

TIME COMPLEXITY:

$O(\sqrt{max(A, B)})$, for each test case.

PREREQUISITES:

Sorting

PROBLEM:

You are given a binary string S.

You are allowed to perform the following operation:

- Select any subsequence of length $2k$
- Move the elements at the first k indices of the sequence to the beginning of the string, and those at the last k indices to the end of the string.

You may perform this operation any number of items. Output the lexicographically minimal string that you can reach.

EXPLANATION:

Note that if $|S| = 2$, then the operations are ineffective, i.e, they do not change S. Therefore, in this case the answer is S itself.

Else, I claim that we can perform operations to sort S completely.

Proof:
If the string consists of only 1's or only $0's$ then we are done. Else, consider the element S_2. If $S_2 = 1$, then we perform the operation with the chosen subsequence being $\{1, 2\}$. This keeps S_1 fixed, and sends $S_2 = 1$ to the end of the string. Else, we perform the operation with the chosen subsequence being $\{2, N\}$. This keeps S_N fixed and sends $S_2 = 0$ to the beginning of the string.

Without loss in generality, assume that $S_1 = 0$ now (if $S_N = 1$ instead, you can perform symmetric operations). For every index i such that $S_i = 1$, you can now perform the operation $\{1, i\}$, which will keep S_1 the same but send $S_i = 1$ to the end of the string. Doing this repeatedly will sort the string.

Therefore, if $N = 2$ then we just output S. Else, we sort S and output that.

TIME COMPLEXITY:

Both $O(N)$ and $O(N \log N)$ are easy.

My solution takes $O(N \log N)$.

PROBLEM:

You are given an array containing $2*\mathrm{N}$ integers.

You can do only the below operation any number of times:

- Choose any a[i] such that a[i] is even and divide it by 2.
- Choose any a[i] and multiply it by 2.

Find the **minimum** number of operation required to make the count of even and odd numbers equal.

EXPLANATION:

Let us first know the minimum number of operations required to change the parity of any element in the given array:

- The minimum number of operations required to convert any odd number in the array to an even number is 1 i.e. by using the second operation once.
- Any even number can be represented as $C.2^B$, where C is odd and $B \geq 1$. The minimum number of operations required to convert any even number $(C.2^B)$ in the array to an odd number is B i.e. by applying the first operation B times so that the new number os C. There is no point in applying the second operation on an even number.

For an even number $X = C.2^B$, we can find out B in $O(log(X))$ time complexity. We will store the B's (for all even numbers in the given array) in an array say D.

Let there be O odd numbers and $E(= 2N - O)$ even numbers in the array. There can be only three possibilities:

- $O = E$, the answer is zero.
- $O > E$, we need to change $(O - E)/2$ odd numbers to even numbers. This will require a minimum of $(O - E)/2$ operations.
- $E > O$, we need to change $(E - O)/2$ even numbers to odd numbers. We will choose the even numbers with the least B's to be changed to odd numbers for the least number of operations. So, the answer is the sum of $(E - O)/2$ minimum elements in array D.

TIME COMPLEXITY:

$O(Nlog(max(A_i))) + O(NlogN)$ for each test case.

PROBLEM:

Given an array A whose elements lie between 1 and M, define the *distance* of an array B (also with elements from 1 to M) to be $\sum_{i=1}^{N} |A_i - B_i|$. Compute the maximum possible distance of an array from A.

EXPLANATION:

It is of course optimal to choose either $B_i = 1$ or $B_i = M$ for each index i.

Note that this choice can be made independently for every index, so the solution is to simply take the best option for each one.

That is, the answer is

$$\sum_{i=1}^{N} \max(|A_i - 1|, |A_i - M|)$$

Note that the answer might not fit inside a 32-bit integer, so make sure to use an appropriate data type.

TIME COMPLEXITY

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

PREREQUISITES:

Prefix Sums, Dynamic Programming

PROBLEM:

Given two matrices A and B, of size $N * M$. The value in each cell denotes the count of students currently in that cell.

- Students represented by matrix A want to reach the left-edge of the field moving only leftwards.
- Students represented by matrix B want to reach the top-edge of the field moving only upwards.

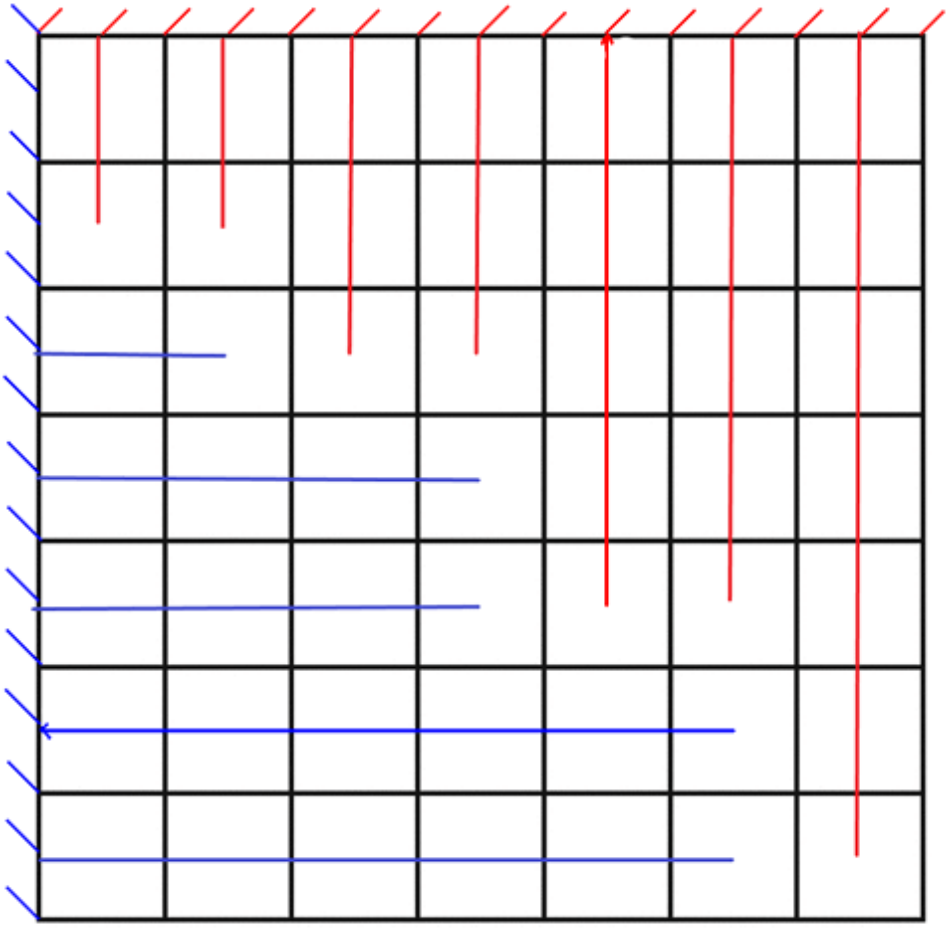
The paths followed by two students going to different hostels should not intersect.

Calculate the maximum number of students which can reach their hostels.

EXPLANATION:

The given problem can be solved by using two-dimensional dp speeded up with prefix sums with $O(M * N)$ time complexity and $O(m * n)$ extra space for the dp matrix.

Now we need to make the partition of the field to maximize the number of students, so this would be step-like structure starting from the top-left end towards the bottom-right end.



Preprocessing: For both matrices A and B, for every column calculate the prefix sums, such that:

- $A[i][j] = A[i][j] + A[i+1][j] ... + A[n-1][j]$
- $B[i][j] = B[0][j] + B[1][j] + ... + B[i][j]$.

Now we would start building our $dp[m+1][n+1]$ matrix from the last column and from bottom to top for every column, and the formula for $dp[row][col]$ is

- $dp[row][col] = max(dp[row+1][col], dp[row][col+1] + A[row-1][col] + B[row][col])$
  (Note: elements going out of the matrix would be equal to 0)

$dp[row][col]$ denote the maximum number of students who could reach their hostels when the submatrix $(row...n) * (col...m)$ is considered.

Now, The maximum number of students that would reach their hostels would be equal to $dp[0][0]$.

TIME COMPLEXITY:

$O(N * M)$ for each test case.

PROBLEM:

You have an array A containing N integers. You are also given Q updates, of the following form:

- L R: add 1 to A_L, -1 to A_{L+1}, 1 to A_{L+2}, and so on

Find the sum of A after all Q updates.

EXPLANATION:

Directly performing an update takes $\mathrm{O}(N)$ time, so performing every update would be $\mathrm{O}(NQ)$, which is too slow.

Instead, let's make a couple of observations about the process.
Let S denote the sum of A. How does S change after an update?

- If the length of the segment we update is even, then S changes by $1 + (-1) + 1 + (-1) + \ldots + 1 + (-1) = 0$
- If the segment has odd length, S changes by $1 + (-1) + \ldots + 1 + (-1) + 1 = 1$

This gives us a rather simple solution that runs in $\mathrm{O}(N + Q)$:

- Compute S, the initial sum of array A.
- Then, for each update:
  - If the length of the segment $[L, R]$ is even, S doesn't change.
  - Otherwise, S increases by 1.
- Finally, print the value of S.

Note that there are ways to actually simulate all the updates fast enough using lazy segment trees or sweeplines, but they are completely unnecessary to solve this task.

TIME COMPLEXITY

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

PREREQUISITES:

Bitwise operations, Binary Search

PROBLEM:

You are given two integers N $(N \geq 2)$ and S. You have to construct an array A containing N integers such that:

- $0 \leq A_i \leq S$
- $A_1 + A_2 + \ldots + A_N = S$
- A_1 & A_2 & .... & $A_N = 0$, where & denotes **bitwise AND** operator.
- The maximum element of the array is minimized.

Find the maximum element of the array A.

EXPLANATION:

Hint 1

For a fixed maximum element x, what is the maximum sum of array elements that can be obtained? Remember, a bit can be set in atmost $N-1$ integers to obtain A_1 & A_2 & .... & $A_N = 0$.

Hint 2

For a fixed maximum element x, say the maximum sum of array elements obtained is S. Then we can create any valid array with $\sum_{i=1}^{N} A_i \leq S$.

Hint 3

Try to think of binary search.

Solution

Let x is the maximum array element of array A of length N and $F(x, N)$ denotes the maximum sum of the array elements that can be obtained satisfing A_1 & A_2 & .... & $A_N = 0$.

How to calculate the value of F(x, N)?

Say $N = 4, x = (101001101)_2$, Then one possible way to obtain the maximum sum of array element is:
Index of bits: 876543210

$A_1 : (101001000)_2$
$A_2 : (101000111)_2$
$A_3 : (100111111)_2$
$A_4 : (011111111)_2$

We set the bits from the most significant bit to the least significant bit. We have to unset a fixed bit in atleast one of the integers of the array to obtain a bitwise AND equal to 0.

- First we set the 8^{th} bit in A_1, A_2, A_3 and unset in A_4. Also, we set the remaining bits of A_4(i.e from 7^{th} bit to 0^{th} bit).
- Now the 7^{th} bit is already set in A_4. We can't set this bit in any other integer, as it would result in numbers greater than the maximum element of the array.
- The 6^{th} is already set in A_4. Now we set this bit in A_1, A_2 and unset in A_3. Also, we set the remaining bits of A_3(i.e from 5^{th} bit to 0^{th} bit).
- Now the 5^{th} bit is set in A_3, A_4, we can't set this bit in any more integer, as it would result in numbers greater than the maximum element of the array.

…

…
(We repeat this process for the remaining bits)

Say, the i^{th} bit can be set in atmost K integers. Now from the above observation, we can see that for every bit two cases occur:

- If the i^{th} bit is set in the x, then $K = N - 1$.
- Otherwise, $K = \min(N - 1$, the number of set bits in x on the left of i^{th} bit).

Thus $F(x, N) = \sum_{i=0}^{i=30}(K \cdot 2^i)$.

How to create a valid array of length N with a sum S which is less than F(x, N)?

Traverse from bit $i = 30$ to $i = 0$: Say the i^{th} bit can be set in atmost K integers. Now two cases may occur:

- $S \geq K \cdot 2^i$: Set the i^{th} bit set in K integers, do $S = S - K \cdot 2^i$ and continue with the remaining bits.
- $S < K \cdot 2^i$: Say $M = \lfloor \frac{S}{2^i} \rfloor$, now set the i^{th} bit set in M integers and do $S = S - M \cdot 2^i$. It is guaranteed that $K > M$. So we could have set the i^{th} bit in $(K - M)$ different integers and the remaining value of $S < 2^i$. Hence we add the remaining S to one of these $(K - M)$ integers. Now the sum of array elements is equal to S, hence we can terminate the loop.

How does binary search work?

It is obvious that if $x_1 < x_2$, then $F(x_1, N) < F(x_2, N)$. So we can apply binary search over the value of x and find the minimum x with $F(x, N) \geq S$.

Further optimization

Make sure you have gone through the above solution.

A bit can be set in atmost $(N - 1)$ integers. So if we want to construct a valid array considering from 0^{th} bit to i^{th} bit, the maximum sum that can be obtained is
$= (N - 1) \cdot \sum_{k=0}^{k=i} 2^k$. So first find the minimum value of i such that $(N - 1) \cdot \sum_{k=0}^{k=i} 2^k \geq S$.

Say the minimum possible value of the maximum element is X. Initially take $X = 0$, so the maximum sum of elements that can be obtained, denoted by S' should also be 0. Take another variable $cnt = 0$. Now traverse from bit $j = i$ to $j = 0$, for every bit first check:

Can we unset the j-th bit in X?

What is the maximum sum of array elements that can be obtained if the j^{th} bit is not set in X?

Here cnt denotes the number of bits from i^{th} bit to $(j + 1)^{th}$ which we have set in X. If we unset the j^{th} bit in X, then the j^{th} bit can be set atmost $\min(N - 1, cnt)$ elements of A, the bits after the j^{th} bit (i.e from $(j - 1)^{th}$ bit to 0^{th} bit) can be set in in atmost $(N - 1)$ integers. So the maximum sum obtained by unsettling j^{th} bit in X is $S' + \min(N - 1, cnt) \cdot 2^j + (N - 1) \cdot \sum_{k=0}^{j-1} 2^k$.

If this value is greater or equal to S, we can unset the j^{th} bit in X and increment the value of S' with $\min(N - 1, cnt) \cdot 2^j$.

Otherwise.....

We have to set the j^{th} bit in X and set j^{th} bit in $(N - 1)$ integers of A, so we increase the value of S' by $(N - 1) \cdot 2^j$.and increment the value of cnt by 1.

TIME COMPLEXITY:

$O(\log S)$ or $O(\log^2 S)$ for each test case

PREREQUISITES:

Observation, prefix sums

PROBLEM:

You have a binary string S. You can swap any two of its elements for a cost of 1. At most once, you can also reverse one prefix of S for no cost. Find the minimum cost of obtaining the lexicographically smallest string from S.

EXPLANATION

Suppose S has x zeros and y ones. Obviously, the smallest string we can obtain is $\underbrace{000\ldots000}_{x \text{ zeros}}\underbrace{111\ldots111}_{y \text{ ones}}$, so we want to calculate the cost of reaching this.

Note that the operations can always be reordered so that the prefix reverse operation is performed first - if any swaps are made before the reverse operation, we could have just as well performed these swaps (on maybe different positions) after reversing the prefix.

So, suppose we fix the length of the prefix we are going to reverse, say i ($0 \leq i \leq N$). We would like to calculate the minimum number of swaps required to sort the string now.

This can be done as follows:

- Look at the first x characters. We want all of them to be 0-s, since doing this automatically ensures the last y characters are 1-s.
- If any of the first x characters is already a zero, it is in place and nothing needs to be done.
- If any of the first x characters is a one, it needs to be swapped out with a zero. This can be done in exactly one move, since each one among the first x characters will correspond to a one among the last y characters.
- So, the number of swaps required is exactly the number of ones among the first x characters! All that remains is to be able to calculate this quickly.

This leads us to the following:

- If $0 \leq i \leq x$, the number of ones in the prefix of length x doesn't change (only their positions may change, the number remains the same).
- If $x < i \leq N$, the length x prefix of the string after reversing upto i consists of the characters $i, i-1, i-2, \ldots, i-x+1$.

So, if we compute the prefix sum of the number of ones in the string, each case can be computed in $\mathrm{O}(1)$, after which we can take the minimum over them all. This gives us an $\mathrm{O}(N)$ solution to the problem.

TIME COMPLEXITY:

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

PROBLEM:

Given a binary string S, in one move you can pick two indices and flip their values. Is it possible to make the final string a palindrome?

EXPLANATION:

You might notice that performing an operation on two indices with two different values essentially swaps the values at these indices. This is already enough to make S into a palindrome whenever:

- N is odd; or
- N is even and contains both an even number of 0's and 1's

Proof

When N is even, for S to be a palindrome both the 0's and the 1's must occur an even number of times, since each $0/1$ is paired with another $0/1$.

It's fairly easy to see that if this condition holds, then just swapping 0's with 1's can make S into a palindrome. For example, if there are x occurrences of 0 and y of 1, then make the first $x/2$ and last $x/2$ characters of S 0 using swaps. This is a palindrome.

Further, any odd-length string can always be made a palindrome.
Either the 0's or the 1's must occur an odd number of times in an odd-length string; suppose it's the 0's.
Use one swap to make a 0 the middle character. Now, the remaining part of the string needs to be an even palindrome, and we satisfy the condition above so we're done.

As it turns out, these conditions are both necessary and sufficient.

Proof

Note that the given operation doesn't actually change the parities of the 0's and 1's.
Let c_0 and c_1 be the respective counts. Then, one operation can change them as follows:

- $c_0 \to c_0 + 2$ and $c_1 \to c_1 - 2$ (flipping two 1's to 0)
- $c_0 \to c_0 - 2$ and $c_1 \to c_1 + 2$ (flipping two 0's to 1)
- Leave c_0 and c_1 unchanged (swapping a 0 and a 1)

All three cases don't change the parities. However, our earlier conditions only depended on the parities of c_0 and c_1. So, those cases are both necessary and sufficient.

Thus, the final solution is as follows: compute c_0 and c_1, the counts of 0 and 1 in the string. Then,

- If both c_0 and c_1 are odd, the answer is "No".
- Otherwise, the answer is "Yes".

TIME COMPLEXITY

$O(N)$ per test case.

PREREQUISITES:

A bit of observation, (optional) a range GCD structure

PROBLEM:

You are given an array A. In one move, you can pick an index i and set $A_{i+1} = \gcd(A_i, A_{i+1})$.
Is it possible to sort A by using this move several times?

EXPLANATION

The solution of this task hinges on the following fact:

Let B_i be the final value at index i after some operations have been applied. Then, $B_i = \gcd(A_i, A_{i-1}, A_{i-2}, \ldots, A_j)$ for some $j \leq i$.

Proof

This can be proved inductively.

For $i = 1$, the result is obvious since no operation can change A_1, so $B_1 = A_1$ always.
Now, consider $i > 1$.
If we don't apply any operation to this position, $B_i = A_i$.
Otherwise, consider the last time an operation changed position i.
B_i is set to the gcd of A_i and whatever the current value at position $i - 1$ is.

But, by the inductive hypothesis, the current value at position $i - 1$ is itself some range gcd, of some range ending at $i - 1$.
So, when we take its gcd with A_i, we simply get a range gcd ending at position i, proving our claim.

Now, how do we use this fact?

Consider the following:

- There is no point changing the value of A_N: it might as well be as large as possible.
- Now let's look at A_{N-1}. We want $A_{N-1} \leq A_N$. To achieve this, we have to perform some operations.
- From our earlier claim, the only thing we can do is replace A_{N-1} with some range gcd that ends at $N - 1$.
- Now it's obvious what we should do: choose this range gcd to be as large as possible, while still remaining $\leq A_N$

This will form the crux of the solution:

- Iterate across the array in decreasing order. When at position i, we will perform operations to ensure $A_i \leq A_{i+1}$.
- To do this, find the largest index j such that $j \leq i$ and $\gcd(A_j, A_{j+1}, \ldots, A_i) \leq A_{i+1}$, and perform these operations.
- Continue on to $i - 1$.

If the array is sorted at the end of this, we are done. Otherwise, there is no way to sort the array.

Now for the implementation: there's an easy way that seems slow at first, and a less easy way using data structures whose runtime is easy to prove.

I like easy implementations!

You can just brute force!
That is, at each index i, start at $j = i$ and keep decreasing j while maintaining the current $\gcd$ till you reach a

point where the $\gcd$ falls below A_{i+1}.
Then, perform operations from index j to index $i-1$ in order, and continue on to $i-1$.

This might look like $O(N^2)$ at first glance, but it isn't: it amortizes to something like $O(N \log M)$ where M is the maximum element of the array, here $M \leq 10^9$.

Proof

Note that an operation can change the value at a given index at most $\log A_i$ times: each time the value changes, it turns into a proper divisor of itself, and so at least halves.

Every operation we perform during our bruteforce *will* change the value at the corresponding index: if we perform an operation that doesn't change the current index, there would be no point to moving the left pointer j past this position anyway so we would not have done this.

So, the total number of operations we perform is bounded above by $O(N \log M)$.
Now, note that each such operation is a gcd operation and hence itself takes $O(\log M)$ time, making our total complexity $O(N \log^2 M)$.

However, taking the running gcd of N integers isn't exactly a complexity of $O(N \log M)$: as can be seen in **this blog**, it's actually $O(N + \log M)$.
A similar reasoning applies here, making our complexity $O(N \log M + \log M) = O(N \log M)$.
I like obvious proofs!

Let p denote the position of the left pointer. Let's bruteforce in $O(N)$ to find the position of p for index $N-1$.

Now, note that A_{N-2} is currently $\gcd(A_p, A_{p+1}, \ldots, A_{N-2})$.
In particular, the optimal position for $N-2$ is always going to be $\leq p$, so it's enough to start the pointer for $N-2$ at p.

The only issue is that, it's easy to calculate the gcd of a bunch of elements when you add a new one, but it's not that easy when deleting an element. That is, knowing $\gcd(A_p, A_{p-1}, \ldots, A_{N-1})$ doesn't really give us enough information to compute $\gcd(A_p, A_{p-1}, \ldots, A_{N-2})$, which is what we need to start the pointer at p.

However, note that this is just a range GCD, and can be computed in a number of ways: using a segment tree, or a sparse table, or even **precomputing all subarray GCDs** and binary searching on this list.

The two-pointer part clearly runs in $O(N)$, and the data structure part adds an extra $O(1) \sim O(\log N \log M)$ depending on implementation, so this is certainly fast enough.

TIME COMPLEXITY

$O(N \log M)$ per test case.

PROBLEM:

Chef considers an array *good* if it has no subarray of size less than N such that the **GCD** of all the elements of the subarray is **equal** to 1.

Chef has an array A of size N with him. He wants to convert it into a *good* array by applying a specific operation. In one operation, Chef can choose any subarray and **reverse** it.

Find the **minimum** number of operations required to convert the array into a *good* array and the operations themselves. If there are multiple ways to achieve the result, print any.
If it is not possible to convert the array into a *good* array, print -1 instead.

Note: A subarray of an array is formed by deletion of several (possibly zero) elements from the beginning and several (possibly zero) elements from the end of the array.

EXPLANATION:

$Observation\ 1$: Suppose the $gcd(A_1, A_2....A_{n-1}) \neq 1$ and $gcd(A_2, A_2....A_n) \neq 1$, then our array would be good.
This is because if gcd of an array is greater than 1, then any subarray of it would have gcd greater than 1.

First thing we would do is precompute the suffix gcd and prefix gcd, such that

$$prefix[i] = gcd(A_1, A_2....A_i)$$
$$suffix[i] = gcd(A_i, A_{i+1}, ...A_n)$$

Now we would loop through the array and at any element, say A_i, we would check if

$$gcd(prefix[i-1], suffix[i+1]) \neq 1.....(i)$$

If it satisfies we would either reverse the array $A[1, 2....i]$ or array $A[i+1, i+2....n]$ with aim to satisfy the initial condition given in $Observation\ 1$.
Now we can do this in atmost two operations. Initially we would check if $prefix[n-1] \neq 1$ or $suffix[2] \neq 1$. If both of these satisfies then no further operation is needed otherwise if any one of them satisfies then we would need one more operation otherwise if none satisfies we would need to do two operations.

TIME COMPLEXITY:

$O(N)$, for each test case

PREREQUISITES:

Binary Search

PROBLEM:

There is a hidden array containing only the integers 1 and 2. You want to find a subarray of this array with sum K. To achieve this, you can ask queries for the sum of any subarray.

Achieve the goal using at most 40 queries.

EXPLANATION:

The small number of queries (and the fact that this is an interactive task) should immediately lead you to look at binary search. But what to binary search on?

I will use $sum(L, R)$ to denote the sum of the subarray $A[L \dots R]$.

Suppose we fix the left endpoint of the subarray, L. Binary search allows us to find the largest index R such that $sum(L, R) \leq K$.
The input guarantees that a subarray with sum K exists, so if we repeat this for every index L, we will eventually find the answer. However, this requires $\mathrm{O}(N \log N)$ queries, which is of course way too much.

Instead, let's analyze some simpler cases a bit. First, fix $L = 1$ and find the appropriate R. Now, there are a couple of cases:

- First, if $sum(1, R) = K$, we are done and can immediately report the answer.
- Otherwise, the only possibility is $sum(1, R) = K - 1$, and $A_{R+1} = 2$. (Do you see why?)
- If $A_1 = 1$, it's easy to see that $sum(2, R + 1) = K$ and we would be done. However, if $A_1 = 2$ we get no further information.

Note that the above results apply more generally to any index L such that $sum(L, N) \geq K$.
For any such index, if we find the largest R such that $sum(L, R) \leq K$, then as long as $A_L = 1$ either (L, R) or $(L + 1, R + 1)$ is the answer.

This tells us that we need to find a 1 in the array: ideally, the leftmost 1.
Finding the leftmost 1 can be done easily with binary search: find the first position p such that $sum(1, p) \neq 2p$.

Now that we have our 1 at position p, if $sum(p, N) \geq K$ we can solve the problem with another binary search, as described above. However, this need not always be the case: what happens when $sum(p, N) < K$?

The important thing to note in this case is that everything to the left of p is a 2. So,

- If K and $sum(p, N)$ have the same parity, simply extend the subarray $[p, N]$ to the left by as much as needed to make the sums equal. This can be done without any queries at all, since only the values of K and $sum(p, N)$ need to be known.
- Otherwise, note that there is no choice for the answer subarray but to exclude the *last* 1 in the array: this is the only way to change the parity of the sum.

To deal with the second case, once again we binary search to find the position of the last 1 in the array: say this is q.
Then, $sum(p, q - 1)$ and K have the same parity, so we can extend p to the left by adding 2's till we get a sum of K.

Note that no matter how the problem is solved, we use at most two binary searches: one to find p, and then either one to find q, or one to find R (such that $sum(p, R) \leq K$).

This gives us $2 \cdot \log 10^5 \approx 34$ queries, along with a constant number more for any checks needed inbetween. My implementation linked below uses 35 queries in the worst case, so the limit at 40 is slightly lenient.

PREREQUISITES:

Prime Factorization

PROBLEM:

After solving some Graph Theory related problems KBG challenged Sameer to solve following problem.

You are given an integer N and Q queries. There is an undirected weighted edge between x and y for following conditions:

- $1 \leq x \leq N$.
- y is divisor of x and weight of this edge will be x divide by y (x/y).

Each query is of the form:

- u v: Given two integers u and v $(1 \leq u, v \leq N)$,

There is no multiple edges and self loops in graph.

For each query, find the **minimum** possible cost to reach v from u.

EXPLANATION:

We can always write v as $u \times A/B$, where A and B are integers > 1 (there are no self loops).

So, it is clear that there always exist a path of length at most 2 from u to v $(u \rightarrow (u \times A) \rightarrow (v = u \times A/B))$. The weight of this path is $A + B$.

For the minimum the path weight, there should not be any common factor of A and B $(GCD(A, B) = 1)$ because in that case we can always divide A and B by their common factor to reduce the path weight. This leads to a contradiction!

We know that $X \times Y \geq X + Y$ for $X, Y \geq 2$. This can be easily proved by taking the larger among X and Y to the left side of the inequality. Using this fact it is clear that if A or B can be broken into any 2 factors ≥ 2 then we will reduce the path weight. In other words, if $A = P \times Q$, where $P, Q \geq 2$ then $P + Q \leq A$.

Hence, for the minimum path weight we will have to keep on breaking A and B into factors (≥ 2) so that no further breaking is possible. This clearly implies that we need prime factorization because primes cannot be broken further.

We can find out the prime factorization of u and v. Let $u = 2^{u_1} \times 3^{u_2} \times 5^{u_3} \ldots$ and $v = 2^{v_1} \times 3^{v_2} \times 5^{v_3} \ldots$ then the combined A/B factor can be written as $2^{v_1 - u_1} \times 3^{v_2 - u_2} \times 5^{v_3 - u_3} \ldots$

The path weight for this would be $2 \times abs(v_1 - u_1) + 3 \times abs(v_2 - u_2) + 5 \times abs(v_3 - u_3) + \ldots$ because the weights add up for A and B as it is. This would be the minimum one for the reasons stated above.

This can be found out using prime factorization using sieve and a hash map.

TIME COMPLEXITY:

$O(log(U)) + O(log(V))$ for each test case and $O(Nlog(log(N)))$ for precomputation.

PROBLEM:

Given a binary string S, find the number of indices such that flipping the value at this index makes S have an equal number of 01 and 10 substrings.

EXPLANATION:

Suppose there are x occurrences of 01 and y of 10 as substrings in S.
We'd like to count the number of flips that makes $x = y$.

First, let's analyze how the given operation affects x and y.

Answer

First, consider flipping $1 < i < N$.

- If $S_{i-1} \neq S_{i+1}$, then there is no change in either x or y.
- If $S_{i-1} = S_{i+1}$, then
  - If $S_i = S_{i-1}$, then x and y both increase by 1
  - If $S_i \neq S_{i-1}$, then x and y both decrease by 1

You might notice that, in any case, $x - y$ doesn't change.
So, flipping a 'middle' index can give us $x = y$ if and only if $x = y$ initially: and if this is the case, then flipping any of the middle indices will give us a good binary string.

That leaves only the endpoints of the strings. Flipping those affects x or y by 1 — the exact change can be caseworked since it only depends on (S_1, S_2) and (S_{N-1}, S_N).

This already gives us a fast enough solution:

- First, compute x and y as described above. This is easy to do in $\mathrm{O}(N)$.
- If $x = y$, then the answer is $N - 2$.
- Then, there are only two more cases: flipping the first and the last character. Simply perform both flips and check whether they make the resulting string good in $\mathrm{O}(N)$ each.

However, there's an even simpler implementation.
The key observation is to notice that, if $S_1 = S_N$, then we will have $x = y$; and otherwise x and y will differ by exactly 1.

Why?

This is not hard to see: the idea is that essentially, 'blocks' of the same character can be treated as a single character.

So, suppose $S_1 = 0$. Then, note that you will alternately encounter 01 and 10 as substrings.
Thus, if $S_1 = S_N$, both occur an equal number of times; otherwise 01 occurs one more time than 10.

This observation leads us to a very simple solution:

- If $S_1 = S_N$, the answer is $N - 2$: flip any of the middle characters.
- Otherwise, the answer is always 2: flipping either of the end characters will make $S_1 = S_N$ and hence $x = y$.

TIME COMPLEXITY

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

PREREQUISITES:

Greedy Algorithms, Maths

PROBLEM:

Chef has an array A of length N.

He calls an index i $(1 \leq i \leq N)$ *good* if there exists some $j \neq i$ such that $A_i = A_j$.

Chef can perform the following operation **at most once**:

- Choose any **subsequence** of the array A and add any **positive** integer to all the elements of the chosen subsequence.

Determine the **maximum** number of good indices Chef can get.

EXPLANATION:

Let $freq[i]$ denote the number of occurrence of i in array A. Suppose we choose an integer d to be added in the operation. Using this operation a number x can be converted y only if $y - x = d$. So, let us divide the numbers in the range $[1, 1000]$ into d sequences of form $[i, i + d, i + 2 * d, ..]$ for all i in the range $[1, d]$.

Now consider frequency array of each of these d sequences i.e., $F_i = [freq[i], freq[i + d], freq[i + 2 * d]...]$. We can add any elements of F_i to answer if it's value is more than 1, i.e. there are more than one occurrence of the element corresponding to that frequency. So, we will try to make each of values in F_i not equal to 1. Also, the above operation can be considered as subtracting some value z at index j $(z \leq F_i[j])$ and adding z at index $i + 1$.

Consider all the subarray X of F_i, such that there are no elements equal to 0 in it and its length is maximum possible. Following cases occur-

- Length of X is even, i.e. $X = [x_1, x_2, .., x_{2k}]$. In this case we can simply add all values of X to the answer. This is because we can convert the X array into $[0, x_1 + x_2, ..., 0, x_{(2k-1)} + x_{2k}]$ using the above operation. We can see that each of the elements of X is not equal to 1.
- Length of X is odd and atleast one element in X is greater than 1. Let $x_j > 1$, then we can think of X as $[x_1, x_2, ..., x_j - 1, 1, ..]$. So, now the length of X is even an using the first case, we can add all values of X to the answer.
- Length of X is odd and all the values in X are equal to 1. In this case, we can simply not consider the x_1, and we are left with an array of even length and using the first case, we can simply add all values of X except 1 to the answer. It can be proved that we cannot make all the indices good in this case.

Finally, we can iterate over values of d from 1 to 1000 and answer will be the maximum number of good indices over all the values of d.

TIME COMPLEXITY:

$O(N * max(A_i))$ for each test case.

PREREQUISITES:

Dynamic programming

PROBLEM:

There is a grid consisting of distinct integers. Two players A and B play a game on it, taking turns moving a token from $(1, 1)$ to (N, M) one step right/down at a time. B moves first.

B can make any available move, while A will always move to whichever neighbor has the higher value.
Let K be the maximum value encountered on the path. If B plays optimally, what is the minimum possible value of K?

EXPLANATION:

First, note that which cell of the grid we are on uniquely defines whose turn it is. For cell (i,j), if $(i+j)$ is even then it is Becky's turn, and if $(i+j)$ is odd it is Anya's turn.

With this in hand, the task can be solved using dynamic programming.

Let $dp(i,j)$ denote the minimum possible value of K if the token starts at cell (i,j) and moves towards (N, M). Our answer is $dp(1, 1)$.

Then,

- If it is Anya's turn, $dp(i,j) = \max(A_{i,j}, dp(next))$, where $next$ denotes the unique cell Anya will move to from (i,j), i.e,
  - $next = (i + 1, j)$ if $A_{i+1,j} > A_{i,j+1}$
  - $next = (i, j + 1)$ otherwise.
- If it is Becky's turn, $dp(i,j) = \max(A_{i,j}, \min(dp(i + 1, j), dp(i, j + 1)))$

The reasons for these transitions should be obvious:

- On Anya's turn, she will definitely move to $next$. The maximum of the path from (i,j) is thus either $A_{i,j}$ or the maximum of the path from $next$.
- On Becky's turn, the maximum of the path is obviously at least $A_{i,j}$. Beyond that, she will choose whichever move minimizes the maximum. There are only two possible moves so we just take their minimum.

This can either be implemented as a recursive function with memoization (with the base case being $dp(N, M) = A_{N,M}$, or an iterative dp where you simply iterate across the grid from bottom to top, left to right. Either way, the complexity is $O(N \cdot M)$.

TIME COMPLEXITY

$O(N \cdot M)$ per test case.

PREREQUISITES:

Basic Math

PROBLEM:

You have a grid with N rows and M columns. You have two types of dominos - one of dimensions $2 * 2$ and the other of dimensions $1 * 1$. You want to cover the grid using these two types of dominos in such a way that :

- Each cell of the grid is covered by exactly one domino.
- The number of $1 * 1$ dominos used is minimized.

Find the **minimum** number of $1 * 1$ dominos you have to use to fill the grid.

EXPLANATION:

The minimum number $1 * 1$ dominos can be found by finding the maximum number of $2 * 2$ dominos that can be placed in the grid. This is because the total area to be covered is constant $(N * M)$.

We know that in each row we can place at most $\lfloor N/2 \rfloor$ number of $2 * 2$ dominos and in each column we can place at most $\lfloor M/2 \rfloor$ number of $2 * 2$ dominos. It is clear that we can always fill a sub-grid of size $(\lfloor N/2 \rfloor * 2) * (\lfloor M/2 \rfloor * 2)$ with $2 * 2$ dominos and this means that the maximum number of $2 * 2$ dominos is $\lfloor N/2 \rfloor * \lfloor M/2 \rfloor$.

Since the total area to be covered is $N * M$ and the maximum area to be covered by $2 * 2$ dominos is $4 * \lfloor N/2 \rfloor * \lfloor M/2 \rfloor$. So the minimum number of $1 * 1$ dominos (= area to be covered by $1 * 1$ dominos) is $N * M - 4 * \lfloor N/2 \rfloor * \lfloor M/2 \rfloor$.

TIME COMPLEXITY:

$O(1)$ for each test case.

PROBLEM:

On an $N \times M$ grid, Alice starts at cell $(1, 1)$ and Bob starts at cell (N, M). Alice moves X steps at a time, and Bob moves Y steps at a time.

Is it possible for Alice and Bob to reach the same cell at the end of Alice's move?

EXPLANATION

Working out a few examples should give you the idea that the only things that really matter are the parities of $N, M, X,$ and Y.

Since $N, M \geq 2$, there are only two cases to consider (and no edge cases):

- The answer is "Yes" if at least one of X and Y has the same parity as $N + M$
- The answer is "No" otherwise

Proof

Let $d = N - 1 + M - 1$.
We claim that the answer is "Yes" If and only if d has the same parity as at least one of $\{X, Y\}$. Note that d has the same parity as $N + M$.

The parity of the manhattan distance between Alice and Bob after the first move is $(d - x) \pmod 2$. After the second move, it is $(d - x - y) \pmod 2$, and so on. This sequence is:
$d, d - x, d - x - y, d - y, d, d - x, \ldots$

The points where it is the end of Alice's turns are $d - x, d - y, d - x, d - y, \ldots$
If the distance is zero after Alice's move, we need this to be 0. So, d should have the same parity as at least one of $\{X, Y\}$. If not, the answer is definitely "No".

Now, we show that this condition is also sufficient.

First, note that if X is odd, Alice can always be made to move exactly one square on her turn. Similarly, Bob can do this when Y is odd.

- **Case** 1**:** If Y is even: We make Bob stay at (N, M) always. Since even X and odd d cannot be the case (since Y is already even), we can always make Alice reach (N, M).
- **Case** $2a$**:** If Y is odd, and X is even, and if $d > X$: We make Alice stay at $(1, 1)$ throughout the beginning, and make Bob reduce the Manhattan distance by 1 in each move. When the distance between them becomes exactly X, Alice moves and meets Bob.
- **Case** $2b$**:** If Y is odd, and X is even, and if $d \leq X$: Either Alice can meet in the first move itself, or Bob reduces the distance by 1, and then Alice meets Bob on the next move (Note that since $N, M \geq 2$, the $d \geq 2$, and so it can be reduced by 1 by Bob without meeting Alice).
- **Case** 3**:** If Y is odd, and X is odd: This means that d is also odd. So, we just make Alice and Bob reduce the distance by 1 at each move, and eventually Alice will be the person to meet Bob in her move.

TIME COMPLEXITY:

$O(1)$ per test case.

PROBLEM

There are N people and the group size preference of i^{th} person is A_i. The objective is to find out if we can assign every person to a group of their preferred size.

EXPLANATION

To solve this problem, let's first try solving a subproblem, which is to find if all the people with a particular value of group size preference can remain happy.

Let's define $happy(r)$ as a boolean, which is $true$ if all people with group size preference r can remain happy, and $false$ otherwise. And define $count(r)$ as the count of people which have group size preference r.

Let $k = count(r)$ and $k > 0$. Then, every one with group size preference r can remain happy only —

- If they can be distributed into groups of size r where each of the k people is in **exactly** 1 group, or
- If $k = r$ or $k = 2 \times r$ or $k = 3 \times r$…, or
- If k is a multiple of r, or
- If $k\%r = 0$

So, if $k > 0$ and $k\%r = 0$, then $happy(r) = true$, otherwise $happy(r) = false$. And we can say that everyone remains happy, if for all values of r from 2 to n (where $count(r) > 0$) $happy(r)$ is $true$.

We're only checking values of r with $count(r) > 0$ as when $count(r) = 0$ there is no sense in saying that people with group size preference r are $happy$ or not as there are no people!

TIME COMPLEXITY

The time complexity is $O(N)$ per test case.

PROBLEM:

Chef is happy with a string if it contains a substring of length strictly larger than 2 consisting of only vowels. Is Chef happy with string S?

EXPLANATION:

If there exists a substring of length more than 2 consisting of only vowels, then there definitely exists a substring of length 3 containing only vowels: for example, just take the first 3 characters of this substring.

So, all we need to do is determine whether some substring of length 3 consists of only vowels. There are $N-2$ such substrings, so we can simply use a loop and check all of them.

That is, run a loop of i from 1 to $N-2$. For each i, check if all 3 of $\{S_i, S_{i+1}, S_{i+2}\}$ are vowels. If this is true for any i, the answer is “Happy”. Otherwise, the answer is “Sad”.

TIME COMPLEXITY

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

PROBLEM:

Chef is teaching his class of N students at Hogwarts. He groups students with the same height together for an activity. Some of the students end up in a groups with only themselves and are saddened by this.

With the help of his magic wand, Chef can **increase** the height of any student to any value he likes. Now Chef wonders, what is the minimum number of students whose height needs to be increased so that there are no sad students?

EXPLANATION:

For all students we first need to find how many students are going to end up in a group with only themselves which is same as counting students with unique heights. Let K be the number of students which have a height H such that count of H in the array is 1.
Now there are four cases :

- $K = 1$ and H is not the maximum height of all the students: The answer is 1 in this case , just increase the height of this particular student to the maximum value of heights of all students.
- $K = 1$ and H is the maximum height of all students but there exists a group of at least 3 students grouped together: The answer is 1 in this case just pick any student from any group with at least 3 students and increase his/her height to the maximum height of all students.
- $K = 1$ and H is the maximum height of all students and all groups consist of 2 students only. In this case the answer cannot be less than 2 as increasing any students height to the maximum height still results in a student who has a unique height. Thus we need to increase the height of at least 2 students to the maximum height of all students in this case.
- $K > 1$: The answer is $ceil((K + 1)/2)$ in this case. If K is even we can form $K/2$ pairs by increasing the height of students.(Sort the students according to height, increase the height of first student to make it equal to second students height and so on from the third student). If K is odd form $(K - 3)/2$ pairs and perform 2 operations to a make of group 3 students (Sort the students according to height, increase the height of first and second student to make it equal to third student's height and then follow the same method for even case).

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Hash Maps

PROBLEM:

Chef has an array A of length N.

Let $F(A)$ denote the maximum frequency of any element present in the array.

For example:

- If $A = [1, 2, 3, 2, 2, 1]$, then $F(A) = 3$ since element 2 has the highest frequency $= 3$.
- If $A = [1, 2, 3, 3, 2, 1]$, then $F(A) = 2$ since highest frequency of any element is 2.

Chef can perform the following operation **at most once**:

- Choose any **subsequence** S of the array such that every element of S is the same, say x. Then, choose an integer y and replace every element in this subsequence with y.

For example, let $A = [1, 2, 2, 1, 2, 2]$. A few examples of the operation that Chef can perform are:

- $[1, 2, 2, 1, 2, 2] \rightarrow [1, 5, 5, 1, 2, 2]$
- $[1, 2, 2, 1, 2, 2] \rightarrow [1, 19, 2, 1, 19, 19]$
- $[1, 2, 2, 1, 2, 2] \rightarrow [2, 2, 2, 1, 2, 2]$

Determine the **minimum** possible value of $F(A)$ Chef can get by performing the given operation at most once.

EXPLANATION:

Let P be the element with the maximum frequency in A (having frequency $freq_P$) and Q be the element with the second maximum frequency in A (having frequency $freq_Q$) . It is possible that Q does not exist. Let us consider these 2 cases:

- Q exists - Initially $F(A) = freq_P$. We should choose $x = P$ because we need to minimize the final value of $F(A)$ as in the question and it is always better to have $y \neq Q$. We should only replace a maximum of $\lfloor \frac{freq_P}{2} \rfloor$ occurrences of P to y because if we replace more, then y becomes the new element with the maximum frequency. Also, replacing less than $\lfloor \frac{freq_P}{2} \rfloor$ occurrences of P to y is not optimal because frequency of P can be further reduced by replacing more occurrences of P and hence $F(A)$ can be reduced further. Hence, it is optimal to replace $\lfloor \frac{freq_P}{2} \rfloor$ occurrences of P to some $y \neq Q$. Now, $freq_P - \lfloor \frac{freq_P}{2} \rfloor$ is the final frequency of P and $freq_Q$ is the final frequency of Q. The maximum among these two i.e. $Max(freq_P - \lfloor \frac{freq_P}{2} \rfloor, freq_Q)$ has to be the answer because frequency of $y = \lfloor \frac{freq_P}{2} \rfloor \leq freq_P - \lfloor \frac{freq_P}{2} \rfloor$ = frequency of P.
- Q does not exist - Everything will remain same like the above point just $freq_Q$ is 0. So, the answer is $freq_P - \lfloor \frac{freq_P}{2} \rfloor$.

This can be implemented very easily using a hash map, which stores the number of occurrences of all elements in A.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

You have an array A containing N integers which are not known to you. You are also given a binary string S of length $n-1$. For each $1 \leq i \leq N-1$, S_i denotes the following:

- If $S_i = 0$, then $A_i < A_{i+1}$
- If $S_i = 1$, then $A_i > A_{i+1}$

For how many values of i $(1 \leq i \leq N)$ is it possible that A_i is the maximum element of the array A.

EXPLANATION:

Insert `0` in the beginning of the string and `1` to the end the string. The number of substring `01` in this modified string is the answer to the problem.
How? For every substring of all $1s$ which have 0 on both there sides(if they exist), only the first number can be assigned the maximum value as this substring represents a **strictly decreasing** subarray. Thus, the number of `01` in the modified string is the answer to the problem

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

Chef has two integers X and Y. Chef wants to perform some operations to make X and Y equal. In one operation, Chef can either:

- set $X := X + 1$
- or set $Y := Y + 2$

Find the minimum number of operations required to make X and Y equal.

EXPLANATION:

There are two cases:
Case 1 $X \leq Y$: Since, Y cannot be decreased by any method, therefore X has to be increased by at least $Y - X$. Thus, this is the minimum number of operations and all operations are of type 1.
Case 2 $X > Y$: Y has to be made at least X. Therefore the minimum number of operations is $ceil(\frac{(Y-X)}{2})$. If X and Y don't have the same parity then Y would be increased to $X + 1$ by doing these operations and thus we need to do one more operation which is increasing X by 1.
$\therefore answer = ((X \leq Y)?Y - X : \frac{(X-Y+1)}{2} + 1 - (X\%2 == Y\%2))$

TIME COMPLEXITY:

$O(1)$ for each test case.

PROBLEM:

You are climbing an infinity staircase, which (as its name suggests) has an infinite number of stairs. You are currently standing at the 1st stair, and you would like to reach the N-th stair. When standing at the i-th stair, you can make one of the following three moves:

- Move to stair $i + 1$
- Move to stair $i + 2$
- Move to stair $i + 3$

However, you can't make the same move twice in a row.

For example, suppose you reach the 7th stair from the 4th stair using the third move. Then, from the 7th stair, you can move to either the 8th or the 9th stair (using the 1st or 2nd move) — but you can't move to the 10th stair (because you would use the third move twice in a row). Now suppose you choose the 1st move and move to the 8th stair. On your next move, you cannot move to the 9th stair, but you can move to either the 10th or the 11th stair.

Under these conditions, find the **minimum** number of moves you need to reach the N-th stair.

EXPLANATION:

Total difference in stairs that needs to be covered is $N - 1$. Given the type of moves and the constraint that you can't make the same move twice in a row makes it impossible to climb more than 5 stairs in 2 consecutive steps ($32323....$). This means the answer is at least $2 \cdot floor(\frac{(N-1)}{5})$ in any case.

- If $N - 1$ is a multiple of 5, the answer is exactly $2 \cdot floor(\frac{(N-1)}{5})$.
- If the remainder when $N - 1$ is divided by 5 is $1, 2$ or 3, it can always be covered in 1 more move, the answer is $2 \cdot floor((\frac{(N-1)}{5})+1$.The answer cannot be less than this because maximum stairs that can be climbed in $2 \cdot floor((\frac{(N-1)}{5})$ moves, in this case ,is only $N - 1 - (N - 1)\%5$.
- If the remainder is 4, exactly two more moves (1 stair then 3 stairs) are needed to cover the remaining stairs, the answer is $2 \cdot floor(\frac{(N-1)}{5})+2$.The answer cannot be less than this because maximum stairs that can be climbed in $2 \cdot floor((\frac{(N-1)}{5})$ moves, in this case is only $(N - 1) - 4$ and in one more move you can only cover at most 3 stairs.

Therefore ,
$ans = ((N - 1)/5) * 2$
$(N - 1)\% = 5$
$if(N - 1)ans+ = (N - 1) <= 3?1 : 2$
$print(ans)$

TIME COMPLEXITY:

$O(1)$ or for each test case.

PROBLEM:

Ashish gave you two positive integers N and K. You have a circular sequence $1, 2, 3, 4, 5.....N-1, N$. That means the number after N is 1.

In one operation you remove the kth element towards right starting from the position of the smallest element in sequence at that time. After $N-1$ operations, only one element remains in the sequence.

Tell the parity of the last element left in the sequence.

EXPLANATION:

There are three cases:

Case 1 $K = 1$: All numbers starting from 1 to $N-1$ will be deleted. N is the last remaining element. The parity of N is the answer is this case.
Case 2 $K = 2$: All numbers starting from 2 to N will be deleted. 1 is the last remaining element. Answer in this case is `ODD`.
Case 3 $K >= 3$:All numbers from K to N will be deleted first. Now numbers from 1 to $K-1$ remain in the list. K^{th} number from 1 is 1. 1 will be deleted. Now numbers from 2 to $K-1$ are present. Now K^{th} number from 2 is 3.
Since the size of the list decreases by 1 after each iteration and number of numbers till the next odd number also decreases by 1. Therefore only odd numbers starting from 1 are deleted till there are no odd numbers left in the list. Hence only an even number must be left in the end. Answer for this case is `EVEN`.

TIME COMPLEXITY:

$O(1)$ for each test case.

PREREQUISITES:

Basic modular arithmetic

PROBLEM:

You have to consider a sequence of numbers where the first element is X. And each element after it, is the sum of all the elements before it. And you need to output the sum of the first N elements of this sequence.

EXPLANATION:

Let us look at the first few elements of this sequence:

The first element is X.
The second element is the sum of everything which came before it, which is just X.
The third element is sum of first two, which is $X + X = 2X$.
The fourth element is sum of first three, which is $X + X + 2X = 4X$.
The fifth element is sum of first four, which is $X + X + 2X + 4X = 8X$.

We see a pattern emerging. From the third element onwards, the elements seem to keep doubling. But it's not obvious why that's the case.

So now, instead, let's look directly at the sum of the first i elements:

The sum of the first element is X.
The sum of the first two elements is $X + X = 2X$.
The sum of the first three elements is $X + X + 2X = 4X$.
The sum of the first four elements is $X + X + 2X + 4X = 8X$.

Here, we again see a very similar pattern, but now, starting from the second sum itself, the sums are double the previous sum.
And this is easy to see why - suppose the sum of the first 5 elements is some Y. Then the 6th element is Y, by definition. So then the sum of the first 6 elements is 2.
So generalizing this, we see that the sums are just $X, 2X, 2^2X, 2^3X, 2^4X, 2^5X, \ldots$. In particular, the sum of the first N elements is just $2^{N-1}X$.

So given X and N, we just have to output $(2^{N-1}X) \mod 1000000007$. Since there are a lot of testcases, and the powers of 2 are independent of X, we can compute $2^i \mod 1000000007$ for all values of i from 0 to 10^6, and store them in an array.

And we use the fact that $(AB) \mod C = ((A \mod C) * (B \mod C)) \mod C$ to use the precomputed values of the modulos of the powers and get the answer for each testcase in constant time after the precomputation.

TIME COMPLEXITY:

Time complexity is $O(N + T)$.

PROBLEM:

N candidates (numbered from 1 to N) join Chef's firm. The first 5 candidates join on the first day, and then, on every subsequent day, the next 5 candidates join in.
For example, if there are 12 candidates, candidates numbered 1 to 5 will join on day 1, candidates numbered 6 to 10 on day 2 and the remaining 2 candidates will join on day 3.

Candidate numbered K decided to turn down his offer and thus, Chef adjusts the position by shifting up all the higher numbered candidates. This leads to a change in the joining day of some of the candidates.

Help Chef determine the number of candidates who will join on a different day than expected.

EXPLANATION:

Total number of groups of 5 would be:

$$total = \lfloor \frac{n+4}{5} \rfloor$$

Total number of groups that won't be affected:

$$unaffected = \lfloor \frac{k+4}{5} \rfloor$$

All the groups that are after the group containing the k_{th} candidate will have 1 person whose joining date will change since he will be shifted to one previous group.
Thus

$$affected = total - unaffected$$

which is our answer.

TIME COMPLEXITY:

$O(1)$, for each test case.

PROBLEM:

You have a binary string S of length N. Find the maximum possible xor of two of its substrings of equal length. The substrings **may** overlap.

EXPLANATION:

First, get a couple of simple edge-cases out of the way: if S consists of only zeros or only ones, the answer is obviously 0, since any two substrings of the same length will be equal.

Now, say S has both 0's and 1's. Let A represent an answer substring, and suppose the substrings chosen to obtain A were $[l_1, r_1]$ and $[l_2, r_2]$.
We can make a couple of observations about A:

- A must begin with a 1. If it doesn't, we can simply take $[l_1 + 1, r_1]$ and $[l_2 + 1, r_2]$ to obtain the same decimal value with a shorter answer string.
- Once we know that A begins with a 1, it's better for A to be as long as possible: any length K binary string starting with 1 represents a strictly larger integer than any length $K - 1$ binary string.

This leads us to a natural 'solution':

- Let L_1 be the position of the first occurrence of a 0, and L_2 be the position of first occurrence of a 1. Without loss of generality, let $L_1 = 1$ and $L_2 > 1$.
- Fix R_1 and R_2 to be as large as possible while maintaining equality of length. In practice, this means that $R_2 = N$ and $R_1 = 1 + N - L_2$.
- Take the xor of the two substrings obtained.

However, this solution is not entirely correct: for example, consider $S = 0001101$. Here, it's optimal to choose $L_1 = 2$ and $L2 = 4$, to obtain the strings 0011 and 1101 for a xor of 1110.

This should give you an idea of what went wrong with the initial solution: choosing L_2 and R_2 was correct, the issue is that L_1 can be anything from 1 to $(L_2 - 1)$: we need to find which one of them is optimal.

This can be done greedily, since we also need to maximize our answer greedily.

Let L_3 be the first position $> L_2$ that contains a 0 (recall that for us, $S_1 = 0$ and $S_{L_2} = 1$. If this is not the case for S, find a 1 instead).
Ideally, when we choose the position of L_1, we'd like it to be such that positions $L_2, L_2 + 1, \ldots, L_3 - 1$ get matched with a 0, and L_3 gets matched with a 1.
The *only* way to do this is to set $L_1 = L_2 - (L_3 - L_2)$.

So, we have a few cases to deal with:

- First, it's possible that L_3 might not exist at all, i.e, there is no 0 after L_2. In this case, note that it's optimal to choose $L_1 = 1$.
- Now, suppose L_3 exists. Set $L_1 = L_2 - (L_3 - L_2)$. We have two cases here:
  - If $L_1 \geq 1$, then nothing more needs to be done: we have our (L_1, R_1) and (L_2, R_2) so we can find the xor of the corresponding substrings.
  - However, it's possible that $L_1 \leq 0$. In this case, note that it's once again optimal to choose $L_1 = 1$.

So, based on these cases, find the values of L_1, R_1, L_2, R_2 and take the xor of the corresponding substrings to obtain A.

Note that the output is the decimal value represented by the answer string A. This can be done easily as follows:

- Initialize the answer ans to 0

- Iterate across the characters of A from left to right
- At each iteration, multiply ans by 2. Then, if the current character is 1, add 1 to ans.
- Remember to always keep ans modulo $10^9 + 7$.

TIME COMPLEXITY

$O(N)$ per test case.

PREREQUISITES:

Lazy Propagation, Segment tree

PROBLEM:

The princess of Shamakhi is trying to find the source of eternal youth so she can restore her lost beauty. It turns out that she needs to collect the tears of a thousand beautiful maidens. To do that, she will accomplish a serious mission with three Bogatyrs, more precisely, update one of their assigned arrays within Q queries, and after each update, she wants to know the power of the Bogatyrs.

More formally, there are three Bogatyrs, each with their own array of N integers — named A, B, and C respectively. The princess of Shamakhi has Q updates of the form (t, pos, val), whose details are as follows:

- If $t = 1$, then assign $A_{pos} = val$
- If $t = 2$, then assign $B_{pos} = val$
- If $t = 3$, then assign $C_{pos} = val$

After every update, you need to find the power of the Bogatyrs (P), which is calculated as follows:

$$P = \max_{1 \le L \le R \le N}(a(L-1) + b(L,R) + c(R+1))$$

Here, the functions a, b, and c are defined as:

- If $1 \le x \le N$, then $a(x) = \sum_{i=1}^{x} A_i$, otherwise $a(x) = 0$.
- if $1 \le x \le y \le N$, then $b(x,y) = \sum_{i=x}^{y} B_i$, otherwise $b(x,y) = 0$.
- if $1 \le x \le N$, then $c(x) = \sum_{i=x}^{N} C_i$, otherwise $c(x) = 0$.

Can you help the princess of Shamakhi to restore her lost beauty by finding the power P of the three Bogatyrs after each update?

EXPLANATION:

Let array P keep the prefix sums for array A, more specifically $P_i = A_1 + ... + A_i$, and array M keep the prefix sums for array B, more specifically $M_i = B_1 + ... + B_i$ and array S keep the suffix sums for array C, more specifically $S_i = C_i + ... + C_N$.
Then the power of the Bogatyrs is the maximum value of $P_{l-1} + (M_r - M_{l-1}) + S_{r+1}$ for some $1 \le l < r \le N$.
Let us simplify the formula, $P_{l-1} + (M_r - M_{l-1}) + S_{r+1} = (P_{l-1} - M_{l-1}) + (M_r + S_{r+1})$.
Let us compute two new arrays D, E, where $D_i = P_i - M_i$ for $1 \le i \le N$, $E_i = M_i + S_{i+1}$ for $1 \le i \le N$. The problem now reduces to maximize the value of $D_l + E_r$ for $1 \le l < r \le N$. This can be done with the help of segment tree, where each node of segment tree keeps the maximum possible value of $D_l + E_r$ (satisfying $l < r$), maximum value of D array in the range, maximum value of E array in the range. Now it remains how to efficiently update these D, E arrays for each type of query.
Updating some position of A array influences the suffix of D. More specifically if we update A_{pos} with val then each value in the suffix D starting from pos is increased by $val - A_{pos}$(decreased if this value is negative).
Updating some position of B array influences the suffix of D and E array.
More specifically if we update B_{pos} with val then each value in the suffix D starting from pos is increased by $B_{pos} - val$ (decreased if this value is negative) and suffix E starting from pos is increased by $val - B_{pos}$ (decreased if this value is negative)
Updating some position of C array influences the prefix of E array. More specifically if we update C_{pos} with val then each value in the prefix of E ending at $pos - 1$ is increased by $val - C_{pos}$(decreased if this value is negative).
As a result, we need to keep two separate lazy propagation arrays for D and E arrays to deal with range addition updates for D and E.

The answer after each query would be the maximum value of $D_L + E_R$ stored in the root node of the segment tree

For details of implementation, please refer to the solutions attached.

TIME COMPLEXITY:

$O((N + Q) \cdot \log(N))$ or for each test case.

PREREQUISITES:

Dynamic programming

PROBLEM:

A permutation is K-beautiful if it can be partitioned into K subarrays such that each subarray consists of elements of the same parity, and cannot be partitioned into $< K$ subarrays satisfying this condition.

You are given a permutation with some of the elements replaced by 0's. How many ways exist to replace the 0's such that the resulting sequence is a K-beautiful permutation?

EXPLANATION:

The main observation here is that every odd number is essentially the same since we only care about parity. That is, it doesn't matter whether we replace a 0 with a 1 or a 3 — only whether we replace it with an even number or an odd number. So, first compute E and O: the total number of unused even and odd integers respectively.

This leads to a dynamic programming solution: we need to keep track of where we are, how many subarrays we have created so far, how many odd/even numbers have been used to replace zeros with so far, and what the parity of the previous element is (to know whether its subarray can be extended or not).

Let $f(pos, ct, ev, od, par)$ denote the number of ways to replace zeros in the first pos positions using od odd integers and ev even integers, such that exactly ct maximal subarrays have been created so far, and the parity of the integer placed at pos is par.

This gives us a somewhat simple recurrence:

- First, suppose $P_{pos} \neq 0$. Then, the parity at this position is fixed, and we don't do any replacements so ev and od do not change. So,
  - If $par \neq parity(P_{pos})$, then $f(pos, ct, ev, od, par) = 0$.
  - Otherwise, $f(pos, ct, ev, od, par) = f(pos - 1, ct, ev, od, par) + f(pos - 1, ct - 1, ev, od, 1 - par)$. That is, if the previous position had the same parity then the number of maximal subarrays (ct) doesn't change, and if it had a different parity the number of subarrays increases by 1.
- Now, suppose $P_{pos} = 0$. There are two cases to consider: replacing this position with an even integer, and replacing it with an odd integer. I'll discuss the odd case, the even one follows similarly.
  - First, note that we have $\max(0, O - od + 1)$ choices for which odd integer to place here (since $od - 1$ were used in previous positions).
  - Then, the recurrence looks similar to the above case: either the parity of the previous position is the same (in which case ct doesn't change), or it is different (in which case ct increases by 1).
  - So, $f(pos, ct, ev, od, 1) = \max(0, O - od + 1) \cdot (f(pos - 1, ct, ev, od - 1, 1) + f(pos - 1, ct - 1, ev, od - 1, 0))$

Our final answer is, of course, $f(N, K, E, O, 0) + f(N, K, E, O, 1)$.

Now, note that we have around $2N^4$ states in our dp. Even with memoization to make transitions $\mathrm{O}(1)$, this results in a $\mathrm{O}(N^4)$ time solution overall, which is too slow.

To speed it up, we can use a somewhat common dp trick: reduce the number of states by looking for a relation between them.

In our case, ev and od are related: note that any relevant state must have $ev + od$ equal to the number of zeros in positions $\leq pos$ (since we *must* replace every zero).

The number of zeros in a prefix is fixed, and thus knowing one of ev or od immediately tells us the other one. So, instead of having both ev and od in our state, it is enough to have only one of them.

This reduces our number of states to $2N^3$, and transitions are still $\mathrm{O}(1)$ so this is fast enough to pass.

TIME COMPLEXITY

$\mathrm{O}(N^3)$ per test case.

PROBLEM:

Given a binary string of length N and an integer K, in one move you can flip any subsequence of the string of length K. How many distinct strings can you obtain using zero or more moves?

EXPLANATION:

The actual binary string S doesn't matter at all — only the values of N and K matter, since each string you can obtain is uniquely determined by the set of positions flipped.

So the question boils down to the following: given N and K, how many subsets of positions can we flip?

The answer has three cases:

- If $N = K$, obviously either we flip nothing or we flip everything, so the answer is 2
- Otherwise, $K < N$, and:
  - If K is odd, we can flip any subset of positions, so the answer is 2^N
  - If K is even, we can flip any subset of positions of *even* size, so the answer is the number of subsets of even size, i.e, 2^{N-1}

Proof

The $N = K$ case is trivial, so let's look at $K < N$.

First, let K be odd. It's enough to show that any single position can be flipped without changing the state of any of the other positions, since this allows us to flip any subset by repeatedly applying it. So, let's show that position 1 can be flipped.

Construction

Consider the positions $1, 2, \ldots, K+1$.
There are $\binom{K}{K-1} = K$ ways to choose a subset of size $K-1$ from positions $2, \ldots, K+1$. To each of these K subsets, insert 1 to obtain a size-K subset.

Now perform K flipping moves, one on each of the subsets we have. Note that:

- Position 1 is flipped K times (an odd number), and so is flipped at the end
- Positions $2, 3, \ldots, K+1$ are flipped $K-1$ times (an even number) each, and so aren't flipped at the end
- Positions $> K+1$ are untouched.

So, only position 1 is flipped, as required.

Next, let K be even. Note that in this case, after each operation the number of positions flipped will be even. It thus remains to show that any even-sized subset is attainable.

Construction

Similar to the odd case, we will show that positions 1 and 2 can be flipped without affecting the rest of the string. Chaining together operations of this type allows any even-sized subset to be formed.

The construction is also essentially the same: there are $K-1$ ways to choose a subset of size $K-2$ from $\{3, 4, \ldots, K+1\}$. To each of these, insert 1 and 2 to obtain a size K subset, and operate on each one.

- Positions 1 and 2 flip $K-1$ times, which is odd.
- Positions 3 to $K+1$ flip $K-2$ times each, which is even

So, at the end only 1 and 2 are flipped and we are done.

Make sure to compute the above values modulo $10^9 + 7$.

PREREQUISITES:

Queues

PROBLEM:

You have a binary string S and an integer K. In one move, you can pick a substring of length K and flip all the values in it. Each substring can be chosen at most once. What is the lexicographically minimum string that can be obtained?

EXPLANATION

For the final string to be lexicographically minimum, we would like as many characters of the prefix to be 0 as possible.

That leads to the following strategy:

- Iterate i from 1 to N
- If $S_i = 0$, nothing needs to be done
- Otherwise, $S_i = 1$.
  - If $i + K - 1 \leq N$, perform the operation on the substring $[i, i + K - 1]$
  - If $i + K - 1 > N$, nothing can be done.

Note that this strategy guarantees that the first $N - K + 1$ characters of the final string will be 0.

Implementing this directly gives us a solution in $\mathrm{O}(N \cdot K)$, which is too slow. However, it can be improved to $\mathrm{O}(N)$ with some observation:

- Suppose we are at position i. We would like to know whether the i-th character is currently 0 or 1 after the previous operations in order to make our decision.
- Note that this depends only on the number of previous operations that affected this index. If an even number of operations affected this index, it will be the same character that it originally was, otherwise it will be the opposite character.

Since we are iterating from 1 to N, this means that at position i, we only care about the operations that might have been made at positions $i - 1, i - 2, \ldots, i - K + 1$ — everything before that definitely doesn't affect this position.

This information can be represented by a queue:

- Maintain a queue of operations performed
- When processing a position i, pop the front of the queue while its value is $< i - K + 1$
- After this, the number of operations affecting i is then exactly the size of the queue.
- Use this to decide whether to perform the operation starting from i or not.
- If the operation is performed, push i into the back of the queue.

This runs in $\mathrm{O}(N)$ time in total.

There are other solutions as well, that do not use queues. You may look at the testers' solutions for these.

TIME COMPLEXITY:

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

PREREQUISITES:

2D arrays, loops

PROBLEM:

Given two cells on the chess board, find if there is a third cell from which a knight can attack both the other cells.

EXPLANATION:

Since the chess board is pretty small - just 8 x 8, we can loop over all the 64 cells and check whether placing a knight there will attack the cells (x_1, y_1) and (x_2, y_2). And even if one of those 64 cells works, we output Yes. If none of them work, we output No.

So now, all that remains is to figure out how to check whether a knight on a call, say (x_4, y_4) attacks a given cell, say (x_3, y_3). For this, from the problem statement, we note that a knight attacks these cells:

```
One square horizontally and two squares vertically away from it, or
One square vertically and two squares horizontally away from it
```

So we just check if x_3 and x_4 are one unit apart, and simultaneously, if y_3 and y_4 are two units apart. If so, then they satisfy the first criteria. To do this, we just check whether the absolute difference of (x_3 - x_4) is 1, and whether absolute difference of (y_3 - y_4) is 2.

Similarly, for the second point, we check whether absolute difference of (x_3 - x_4) is 2, and whether absolute difference of (y_3 - y_4) is 1.

If either of these two points are satisfied, we know that the cell is attacked.

TIME COMPLEXITY:

Time complexity is $O(8 * 8) = O(1)$.

PREREQUISITES:

Familiarity with gcd

PROBLEM:

You have an array A and an integer K. Let $G = \gcd(A)$.
Find out whether there exist K disjoint non-empty subarrays in A with gcd $g_1, \ldots, g_K$ such that $\frac{g_1+g_2+\ldots+g_K}{K} = G$.

EXPLANATION:

First, note that the gcd of any subarray of A will certainly be at least G, since G divides every element of A.
So, each $g_i \geq G$, which means the only way their average can equal G is if each g_i itself equals G.

So, we need to find if K disjoint subarrays in A have gcd G.
Note that since G divides every element of A, if we have a subarray whose gcd is G, extending this subarray to the right or left will still leave its gcd as G.

In particular, if a solution exists, then there will always exist a solution that covers the full array.
This gives us a simple algorithm to check:

- While the array is not empty, find the smallest prefix of the array with gcd G. If no such prefix exists, stop.
- This prefix (if found) will form one subarray in the answer. Remove this prefix and do the same to the remaining array.

The number of times we successfully performed the operation equals the maximum number of disjoint subarrays with gcd G that can be obtained. Check if this number is $\geq K$ or not.

This can be implemented quite easily:

- Keep a variable denoting the current gcd, say g. Initially, $g = 0$.
- For each i from 1 to N:
  - Set $g := \gcd(g, A_i)$.
  - If $g > G$, continue on
  - If $g = G$, increase the answer by 1 and reset g to 0.

TIME COMPLEXITY

$O(N \log(maxA))$ per test case.

PREREQUISITES:

Dynamic programming

PROBLEM:

Given a tree on N nodes where node i has value A_i, for each u from 1 to N compute

$$\sum_{i=1} \lfloor \frac{A_i}{2^{d(u,i)}} \rfloor$$

EXPLANATION:

The main observation here is as follows: if $d(u, i) > 20$, then $\lfloor \frac{A_i}{2^{d(u,i)}} \rfloor = 0$ no matter what the value of A_i is, since $A_i \leq 10^6$.

This means that we only care about vertices at distances ≤ 20 from a given u.
Of course, this doesn't directly solve the problem, but it's a start.

Let's root the tree at some node, say 1.
With this root, let $p^i(u)$ denote the i-th ancestor of u. In particular, $p^0(u) = u$ and $p^1(u)$ is the parent of u.

Note that u contributes a value of $\lfloor \frac{A_u}{2^i} \rfloor$ to $p^i(u)$, and vice versa.

In particular, this allows us to, at least, compute the answer for every u when only considering values that lie in its subtree: for each u, add $\lfloor \frac{A_u}{2^i} \rfloor$ to the answer of $p^i(u)$ for each i from 0 to 20.
This takes $O(20N)$ time.

Now, let's look at a specific u. We've already computed the contribution of things in its subtree, so we need to look outside.
So, let's look at $p^1(u)$. Consider some node v in the subtree of $p^1(u)$, that is *not* in the subtree of u.
If $d(v, p^1(u)) = k$, then $d(v, u) = k + 1$. Can we use this in some way?

Yes, we can!
Let's compute a 3D dynamic programming table: $dp[u][k][x]$ stores the following:

- Consider the subtree of vertex u, and all nodes at a distance of k from u in this subtree.
- $dp[u][k][x]$ holds the contribution of such nodes to a node at a distance of x from u.

So, coming back to our earlier discussion, the contribution of nodes in the subtree of $p^1(u)$ to the answer of u can be contributed using $dp[p^1(u)][k][1]$ across all k.
Note that this will also include some values in the subtree of u, which shouldn't be counted: their contribution can be subtracted out separately using the appropriate cell in the dp table.

Note that this allows us to visit every relevant ancestor of u and do the same thing. That is, for each $0 \leq i \leq 20$, visit $p^i(u)$ and add the values of $dp[p^i(u)][k][i]$ across all k, while also subtracting appropriate dp values to ensure that nothing is double-counted.
This will cover every node that is at a distance of ≤ 20 from u, which is exactly what we wanted.

The algorithm given above takes $O(20^2 \cdot N)$ time and space.
It's possible to optimize the space to $O(20N)$, but this optimization was unnecessary to get AC.

PREREQUISITES:

Dynamic Programming , Permutations and Combinations.

PROBLEM:

You are given an integer array A of size N and a positive integer M. This array contains a hidden message which can be decoded by the following algorithm:

```
int ox[N+1], oy[N+1];
for(int i = 1; i ≤ N; i++) {
    ox[i] = A[i] / M;
    oy[i] = A[i] % M;
}
```

There are N landmines on the xy-plane, such that the i^{th} landmine $(1 \leq i \leq N)$ is located at (ox_i, oy_i).

Initially, you are at point $(0, 0)$ and you must go to point (X, Y). At each step, you can either move to the right by 1 unit or move up by 1 unit.
Formally, if you are standing at (x, y), then you can either move to $(x + 1, y)$ or to $(x, y + 1)$.

A path is called *safe* if there aren't any landmines on that path. Count the number of *safe* paths which you can take to reach the destination. As the answer can be quite large, output it modulo $10^9 + 7$.

EXPLANATION:

Suppose there are no obstacles then number of ways to move from point $(0, 0)$ to say point (x, y) will be $\binom{x+y}{x}$. You can read more about it **here**.

Thus we know the total number of ways to reach point (x, y) which is $\binom{x+y}{x}$. Now we will subtract all the invalid ways, i.e ways that go through a landmine.

To calculate number of invalid ways we go through each cell having a landmine and calculate number of ways to reach at that cell and multiply it with number of ways to reach (x, y) from that cell which would give all the number of ways to reach (x, y) passing through that mined cell. We would add all the invalid ways for each such cells to get the total number of invalid paths. We would simply subtract this from total number of paths to get our answer.

Now in order to calculate number of ways to reach a mined cell, we can use $dp[i][j]$, which tells us the number of ways to reach point (i, j) without stepping on any landmine.

$$dp[i][j] = (dp[i-1][j] + dp[i][j-1]) \ \% \ mod$$

Now if there is a landmine on point (i, j), then

$$invalids \mathrel{+}= (dp[i][j] \times \binom{x-i+y-j}{x-i}) \ \% \ mod$$
$$dp[i][j] = 0$$

Thus finally our answer would be:

$$\binom{x+y}{x} - invalids$$

TIME COMPLEXITY:

$O(Mlog(X+Y)),$

$W = max(A[i]/M) + 1,\ 1 \le i \le N$
$H = max(A[i]\ mod\ M) + 1,\ 1 \le i \le N$
So the following inequality is always true :
$W \le (1000000 + M)/M$ and $H \le M$
which gives
$W \times H \le 1000000 + M$

PREREQUISITES:

Binomial coefficients

PROBLEM:

Alice wants to test Bob's love for the number 7.

Alice gives Bob a sequence of integers $A = [A_1, A_2, \ldots, A_N]$, and Q queries. In each query, she provides an integer K and asks bob to find the number of *lovely* subsequences of size K that can be formed from A.

A sequence B of length K is called *lovely* if, for every subsequence S of B, the following condition holds:

- If $|S|$ is divisible by 7, then the sum of S should also be divisible by 7.
- If $|S|$ is not divisible by 7, then the sum of S should also not be divisible by 7.

where $|S|$ denotes the length of S.

Help Bob answer the queries. Since the answers can be very large, print them modulo $10^9 + 7$.

Note: Two subsequences are considered distinct if the indices chosen to form them are distinct, even if their elements end up being the same. For example, $A = [1, 1, 2, 2]$ has 4 subsequences of length 1 (two are $[1]$ and two are $[2]$) and 6 subsequences of length 2.

EXPLANATION:

As we are concerned with only the remainder of the sum of numbers, for all index i from $i = 1$ to N, replace each element of the array A by $A_i\%7$ and keep the count of all elements in the new array. Let Cnt_i represent the count of i in the modified array A. No index i with $A_i = 0$ can be a part of a *lovely* subsequence as the subsequence formed by taking only this element does follow the second condition.
Now, for $K > 7$ all *lovely* subsequences consist of exaclty 1 distinct number from 1 to 6.

Proof

Let us suppose there exists a *lovely* subsequence of A with length > 7 which contains at least 2 distinct numbers. Select a subsequence, from this *lovely* subsequence, of size 7 such that it contains at least 2 distinct numbers. The sum of this subsequence has to be divisible by 7 as this is a subsequence of size 7 of a *lovely* subsequence (first condition). Now, pick any element from the remaining element of the *lovely* subsequence and swap this with any element in the subsequence, we took, of size 7 which is not equal to it (which must exist as there are two distinct values in the subsequence) and this sum cannot be divisible by 7 now.
$\therefore$ for $K > 7$ any *lovely* subsequence contains no more than 1 distinct number.

For $K > 7$ iterate on values from i=1 to 6 and add $\binom{Cnt_i}{K}$ to the answer.

For $K <= 7$ we can pre-calculate the answer for each value of K from 1 to 7. We have 6 values for each element of the subsequence and this generates a total of $6^1 + 6^2 .....6^7$ different combinations (Since only count of each value is important sort all the combinations later). Iterate on each combination and check whether it is a *lovely* subsequence. The actual total number of lovely subsequences comes out to be very less (102). Now, for each *lovely* subsequence find the number of ways in which it can be constructed from A.
Let C_i represent the count of i in a *lovely* subsequence of size ≤ 7, then the total number of possible ways to construct this *lovely* subsequence from A is given by $\prod_{i=1}^{6} \binom{Cnt_i}{C_i}$

Add this to the answer for a particular size K (equal to the length of this *lovely* subsequence).
Pre-calculations can de done for binomial coefficients to avoid increasing the runtime of the solution.

For details of implementation, please refer to the solutions attached.

TIME COMPLEXITY:

$O(N + Q)$ or for each test case.

PROBLEM:

You have two arrays A and B of length N. In one step, you can add 1 to an element of A and an element of B. Find the minimum number of moves to make A and B equal.

EXPLANATION:

First, note that each move increases the sum of both A and B by exactly one. So, if initially A and B have different sums, they can never be made equal (since they must have the same sum when they are equal).

Now, assume $sum(A) = sum(B)$. Let's look at a specific index i ($1 \leq i \leq N$).

- If $A_i = B_i$, we don't need to do anything
- If $A_i < B_i$, we need to use $B_i - A_i$ operations on A_i to attain equality at this index.
- If $A_i > B_i$, we need to use $A_i - B_i$ operations on B_i to attain equality at this index.

Let $x = \sum_{i=1}^{N} \max(0, B_i - A_i)$. As noted from above, x is the minimum number of operations we need to perform on indices of A.
Similarly, let $y = \sum_{i=1}^{N} \max(0, A_i - B_i)$ be the minimum number of operations we need to perform on B.

Clearly, the answer is at least $\max(x, y)$.

In fact, given the condition that $sum(A) = sum(B)$, we have $x = y$, and this is exactly the answer.

Proof

Note that $x - y = \sum_{i=1}^{N} (B_i - A_i)$, which equals $sum(B) - sum(A)$. Since these are equal by assumption, $x - y = 0$ and hence $x = y$.

Since $x = y$, each of the x operations that need to be made on an element of A can be matched with one of the y operations that need to be made on an element of B, thus completing the proof.

Computing the value of x and/or y can be done in $\mathrm{O}(N)$ with a simple loop.

TIME COMPLEXITY

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

PREREQUISITES:

Depth first search, dynamic programming, sieve of Eratosthenes/some fast prime factorization method.

PROBLEM:

You have an integer N, that you want to make 1. You also have M pairs of the form (a, b), where a is prime. In one move, you can:

- Pick a prime p that divides N
- Divide N by p
- Multiply N by b for all pairs of the form (p, b).

Find out whether N can be made 1, and if it can, the minimum number of moves to do so.

EXPLANATION:

First, note that we can think of this problem entirely in terms of primes. A pair of the form (a, b) essentially removes a from the prime factorization, and adds in all the prime factors of b.

Next, note that there are not actually any choices to be made in terms of number of moves. So, if there is a way to reduce N to 1, then no matter how it is done the cost will be the same.

Thus, we have to find two things:

- Is it possible to reduce N to 1?
- If it is, find the cost of doing so.

Let's look at a specific prime, say p. What is the cost of removing p from the prime factorization?
Well, based on the pairs we have, removing one copy of p will add in some other primes to the factorization. Let these primes be $p_1, p_2, \ldots, p_k$. Then, each of these primes must be removed, and so on.

So, if $cost_p$ denotes the minimum number of moves to remove p from the factorization, we have

$$cost_p = 1 + \sum_{i=1}^{k} cost_{p_i}$$

This can be implemented to run quickly (in $O(V + E)$ where V is the number of vertices and is $\leq 10^6$, and E is the number of edges in the graph) with the help of dynamic programming, since each $cost_p$ only needs to be computed once. The final answer is the sum of $cost_p$ for all primes in the prime factorization of N. For example, for $N = 12$ the answer would be $cost_2 + cost_2 + cost_3$.

The only time we run into issues is if we run into a cycle - that is, if removing p from N ends up with adding p back to N. In this case, bringing N down to 1 is of course impossible, making the answer -1.

Note that you have to be careful about one thing: it is possible that the graph has a cycle somewhere in it, but this cycle is not reachable from any of the primes in the factorization of N. In this case, the answer won't be -1. A trivial example of this is when $N = 1$ and the graph can be whatever, with the answer being 0.

Note that the above solution requires computing the prime factorization of a number quickly. This can be done with the help of the sieve of Eratosthenes as follows:

- Let spf_i denote the smallest prime factor of i
- spf_i can be computed with a sieve
- To factorize x, do the following:

- If $x = 1$, stop
- Otherwise, let $p = spf_x$: p is one factor of x.
- Divide x by p and continue.

N has at most $\log N$ prime factors, so the above process takes $O(\log N)$ time to prime factorize N. Note that this also gives us a bound on the number of edges on in the graph: each of the M pairs adds at most $\log(10^6) = 20$ edges to the graph, so there are $20 \cdot M$ edges at worst.

TIME COMPLEXITY

$O(N \log N)$ per test case, where $N = 10^6$.

PROBLEM:

You have a string S. In one move, you can choose two adjacent equal characters, turn the first one into 0 and delete the second.
Is it possible to bring the string down to a single character?

EXPLANATION:

If $N = 1$, the string is already of length 1 so the answer is “Yes”.

Otherwise, let’s look at what the operation does:

- If there are two zeros in a row, it deletes one of them
- If there are two ones in a row, it essentially deletes both of them and inserts a zero there.

In particular, if we have a continuous block of 1's, then the operation can only reduce its length by 2. So, if this block has odd length, then there will always be a 1 left in the string.

However, note that when $N > 1$, the final character will definitely be a 0.
So, we need to delete every 1 from the string.

This gives us a solution:

- Find every (maximal) continuous block of ones in S.
  - For example, if $S = 0110101110101100 01$, we find blocks of length $(2, 1, 3, 1, 2, 1)$
- If any of these blocks has odd length, it’s not possible to remove it completely and so the answer is “No”.
- If every block has even length, the answer is “Yes”.

TIME COMPLEXITY

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

PROBLEM:

Calling out the main points here that are important to the solution

- We are given a binary string S of length N
- We can remove any element from the string S. We can perform this operation **at most** $\lfloor \frac{N}{2} \rfloor$ times
- The important point is we have to output **any palindrome** that can be obtained in this manner. Kindly note - it can be any palindrome and doesnt specifically have to be a palindrome of the longest length

EXPLANATION:

S is a binary string and has only **0s** and **1s**
The special constrain around removal of $\lfloor \frac{N}{2} \rfloor$ simplifies this problem.
We just need to count the number of 0s and 1s in the original string

- If $Count(1) > Count(0)$ - we can remove all the zeros and print a string containing only 1s
- If $Count(0) \geq Count(1)$ - we can remove all the ones and print a string containing only 0s

TIME COMPLEXITY:

Time complexity is $O(N)$.

PROBLEM:

Given an array A, in one move you can pick any $1 \leq L \leq R \leq N$ and add 1 to all A_i such that $L \leq i \leq R$.
Find the minimum number of moves to make A a palindrome.

EXPLANATION:

We want to make A a palindrome, so let's look at corresponding pairs in A, i.e, pairs of indices $(i, N+1-i)$. If $A_i \leq A_{N+1-i}$, then A_i needs to be increased to reach A_{N+1-i}.

So, let's consider a new array B of length N, where

- $B_i = A_{N+1-i} - A_i$, if $A_i \leq A_{N+1-i}$
- $B_i = 0$ otherwise.

B_i is the *least* number of operations that need to cover index i so that A can be made a palindrome.

In fact, we can always perform operations in such a way that exactly B_i of them touch index i, and it's not hard to see that this is optimal.

Now, all that remains is to minimize the number of operations we perform. That can be done greedily, as follows:

- Let's iterate i from 1 to N.
- When $i = 1$, perform B_1 operations on index 1
- When $i = 2$,
  - If $B_2 \leq B_1$, we can extend B_2 of the operations performed on index 1 to also cover this index, for no extra cost.
  - if $B_2 > B_1$, then we need $B2 - B_1$ *extra* operations starting at index 2.
  - Note that the above two cases can be combined to say that we perform $\max(0, B_2 - B_1)$ operations starting at index 2.
- Now notice that this applies to any index i where $2 \leq i \leq N$, i.e, we need $\max(0, B_i - B_{i-1})$ operations starting at this index.

So, the final answer is simply

$$\sum_{i=2}^{N} \max(0, B_i - B_{i-1})$$

TIME COMPLEXITY

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

PREREQUISITES:

Bitwise Operations

PROBLEM:

You are given 3 numbers A, B and C.

You want to make all these 3 numbers equal. To do this, you can perform any finite number of operations:

- In the i^{th} operation, you must add $2^{(i-1)}$ to any **one** of these 3 numbers.

Find whether you can make these 3 numbers equal in finite number of moves.

EXPLANATION:

Here in each operation we are adding a set bit as we move along from right to left. When we are on the i_{th} operation, i.e adding the i_{th} set bit, we look for all possibilities:

- In case all three numbers are already equal, we can end the operation and conclude that it is possible to make all three numbers equal.
- Else if any two of them have a set bit, we can simply add set bit to the number having the unset bit at i_{th} position, this way all three numbers will have same bit at the i_{th} position and we can move to the next bit position.
- Else if any two of them have unset bit, then we can add set bit to the number having the set bit at i_{th} position, to make it unset at that position. This way all three numbers will have unset bit at the i_{th} position and we can move to the next bit position.
- In any other case, i.e case when we have no set bits or all set bits, there is no way we can make all three numbers to have same bit at the i_{th} position so it won't be possible to make the three numbers equal.

TIME COMPLEXITY:

$O(log(N))$, for each test case.

PREREQUISITES:

Basic combinatorics, **Maximum subarray sum**

PROBLEM:

You have a binary string S. At most once, you can pick a substring of S and flip all its characters.

If the substring is chosen optimally, what is the maximum value of the sum of the count of 1's of each substring, across all substrings?

EXPLANATION:

Let's first forget about the flipping operation and try to quickly compute the value for a given string. Obviously iterating over all substrings would take $\mathrm{O}(N^2)$ time, which is too much.

The quantity we want to calculate is the count of 1's in each substring, summed across all substrings.
Let's look at it from a slightly different perspective: how much does each 1 present in the string 'contribute' to the final answer?

Answer

Suppose $S_i = 1$. S_i then contributes $+1$ to the final answer exactly once for each subarray it is in.

A simple combinatorial argument tells us that the number of subarrays it is present in equals $i \cdot (N - i + 1)$: the left endpoint of the subarray has i choices $(1, 2, 3, \ldots, i)$ and the right has $N - i + 1$ choices $(i, i + 1, i + 2, \ldots, N)$.

So, let's define $B_i = i \cdot (N - i + 1)$. Then, the answer for S is simply the sum of B_i across all those positions i such that $S_i = 1$, which can easily be computed in $\mathrm{O}(N)$.

Now let's look at how flipping can change things:

- If $S_i = 1$, then flipping this position will *decrease* the answer by B_i (since it used to contribute to the sum, and won't after the flip).
- If $S_i = 0$, then flipping this position will *increase* the answer by B_i.

Flipping a substring is then equivalent to adding/subtracting the relevant values of B_i, which is essentially just a subarray sum!
In fact, suppose we define another array C as follows:

- $C_i = +B_i$ if $S_i = 0$
- $C_i = -B_i$ if $S_i = 1$

Then, it's easy to see that flipping the range $[L, R]$ in S changes the answer by exactly the subarray sum of C from L to R.

Of course, we want this change to be as large as possible, since our aim is to maximize the answer. This means we want to find the maximum subarray sum of C, which can be done in $\mathrm{O}(N)$ in a variety of ways.

Thus, the final solution is:

- Compute the B and C arrays as mentioned above.
- Using B, compute the answer for S without flips.
- Then, find the maximum subarray sum of C and add it to the answer.

PREREQUISITES:

Sorting

PROBLEM:

You are given a string consisting of digits from $0-9$, along with $+$ and $-$ characters. You are guaranteed that this string forms a valid mathematical expression.

You can arbitrarily permute the string. Do this in a way such that the final string represents a valid mathematical expression, and its value is maximised.

EXPLANATION:

We want the positive terms to be as large as possible, and the negative terms to be as small as possible.

Assume we have A $+$ characters, and B $-$ characters. Sort the remaining digits in non increasing order. Let's say we have $C = N - A - B$ digits.

Note that the first number in our expression is always positive, so we want to maximise it. We need to leave at least $A + B$ digits for the rest of the expression, we we can use $D = C - A - B$ digits on the first number itself. Now, in order to maximise the number, we should use the largest D digits, so that the largest digits is in the most significant spot. Eg. if we have digits $\{5, 4, 3, 2\}$ and $D = 3$, then we should set the first number to be 543.

Now, each of the remaining numbers will have 1 digit each. A of these digits will be added positively, and B will be added negatively. Therefore, we should create the expression by first alternating $+$'s with the largest A remaining digits, and then alternating $-$'s with the remaining digits.

It's easy to prove that this algorithm is correct, let's see how it works on a sample input.

$N = 7, S = 4 - 89 + 20$

$A = 1, B = 1, D = 3$. the digits are $\{9, 8, 4, 2, 0\}$.

Now, the first number should be 984.

Now, we should alternate A $+$'s with the largest A remaining digits. This gives us $+2$.

Finally, we should alternate B $-$'s with the B remaining digits. This gives us -0.

Therefore, our expression is $987 + 2 - 0$

TIME COMPLEXITY:

$O(N \log N)$

PROBLEM:

Let $f(X)$ denote the number of set bits in X.

Given integers N and K, construct an array A of length N such that:

- $0 \leq A_i \leq K$ for all $(1 \leq i \leq N)$;
- $\sum_{i=1}^{N} A_i = K$

Over all such possible arrays, find the **maximum** value of $\sum_{i=1}^{N} f(A_i)$.

EXPLANATION:

Claim : Each bit that is present in the numbers (except the **most significant bit**) of all the numbers is either present in all N numbers or $N-1$ numbers.

Proof

Let us suppose there exists a bit at position i which is set to 1 in $N-2$ or less numbers and a bit $i+1$ is set to 1 in at least one number in A, then we can unset the bit $i+1$ from a number and set i^{th} bit in two numbers in which it was unset earlier. This increases the total number of set bits without changing the sum of the numbers.

Start iterating from $i = 0$ to $i = 30$ or until $K > 0$. There are three cases now:

- If $K \leq N$ set K bits at the i^{th} position and end the loop.
- If $K > N$ and the parity of N and K is same then some higher bit will be set in the next steps to achieve the sum K. This means i^{th} bit has to be set in either N numbers or $N-1$ numbers. Since the parity is same we have to set it to N numbers.
- If $K > N$ and the parity of N and K is not same then some higher bit will be set in the next steps to achieve the sum K. This means i^{th} bit has to be set in either N numbers or $N-1$ numbers. Since the parity is not same we have to set it to $N-1$ numbers.
  Remove the contribution of each bit after each step from the sum by subtracting the number of bits taken and dividing K by 2.

TIME COMPLEXITY:

$O(log(K))$ for each test case.

PREREQUISITES:

Sieve of Eratosthenes for fast prime factorization

PROBLEM:

You have an array A. In one move, you can replace two adjacent numbers by their $\gcd$ and lcm. What is the maximum possible sum of A after performing some moves?

EXPLANATION:

In tasks that have operations to be performed, it's always nice to make some observations about how things change, and if they change at all.

Say we apply an operation on two integers x and y, with $x \leq y$.

- If x is a factor of y, then $\gcd(x, y) = x$ and $\text{lcm}(x, y) = y$, so there's no change at all
- Otherwise, we know for sure that $\text{lcm}(x, y) \neq y$, which means it must be at least $2y$. In particular, $\gcd(x, y) + \text{lcm}(x, y) \geq 1 + 2y > x + y$, so it is always optimal to perform this move since it improves the sum.
- Also note that the operation allows us to make the final array sorted, since the lcm is always placed after the gcd.

This tells us what the final array will look like in the optimal case: it must be sorted in non-decreasing order, and $A_i \mid A_{i+1}$ for each $1 \leq i < N$.

In fact, under these conditions, the final array that maximizes the sum is unique. The proof of this uniqueness will also allow us to construct the array, and hence compute its sum.

Proof

Let's look at a single prime p. Let pw_i denote the highest power of p that divides A_i.
Note that an operation on A at index i changes pw as follows:

- If $pw_i \leq pw_{i+1}$, do nothing
- Otherwise, swap pw_i with pw_{i+1}

This, of course, allows us to sort the pw array since we have adjacent swaps. Note that the condition $A_i \mid A_{i+1}$ also necessitates that pw be sorted.

$\gcd$ and lcm operate independently on each prime, and so the above discussion applies to each one of them: their respective pw arrays must be sorted in the end.

Knowing the pw arrays also tells us the elements, since we effectively know the prime factorizations of each element.

That gives us a working solution:

- For each prime p, compute its pw array and sort it.
- Compute the final answer from all the pw arrays for all the primes.

However, this by itself is too slow: there are $\approx 8 \cdot 10^4$ primes less than 10^6, and computing the pw array for each of these in $\text{O}(N)$ is much too slow.

Instead, note that most elements of most pw arrays will just be 0. In particular, only $\text{O}(N \log 10^6)$ positions will be non-zero across all arrays, corresponding to the prime factorizations of the input elements.

So, prime factoring the input is enough to get all the information we need, since 0's in the pw array don't affect anything anyway. Keeping this compressed information about non-zero powers allows us to solve the problem quickly.

Thus, the final solution is as follows:

- Prime factorize each A_i quickly.
  - This can be done in $\mathrm{O}(\log 10^6)$ with a sieve of Eratosthenes that precomputes, for each integer, its smallest prime factor.
- Use these prime factorizations to build compressed pw arrays, that don't include the zeros.
- Sort the obtained arrays and give them to positions in reverse order, starting from N
- Finally, compute the sum of the resulting array.

PREREQUISITES:

Minimum Spanning Tree, Kruskal Algorithm

PROBLEM:

Yash gave Swar 2 integers N and M, and an integer array A of size M. Swar has to consider all undirected weighted graphs G such that:

- G has N vertices and M edges
- G is simple, i.e, there is at most one edge between any pair of its vertices and it doesn't contain self-loops
- The weights of the edges of G are exactly the array A

Among all these graphs, Swar has to find the **maximum** possible weight of a minimum spanning tree. Please help Swar find his answer.

OBSERVATION and INTUITION

Let us look at this problem from the view of Kruskal Algorithm: First the edges are going to be sorted in non-decreasing order. Then an edge is taken into **MST** if the nodes it connects lie in different connected components otherwise it is skipped. The algorithm stops when $N-1$ edges are taken.
This means the minimum weight edge is going to be taken in the **MST** as it connects two different components. Let the nodes added in the **MST** be $1->2$ Now since multiple edges between two nodes are absent, second edge also connects two different components and hence has to be taken into the **MST**. Let the graph now be $(1->2->3)$. We construct this graph instead of $1->2$ and $3->4$ .so that we can skip the third edge otherwise it will again connect two distinct components. Now we can construct a graph in which the third edge connects the the first node and the third node and thus it can be skipped. This is the maximum number of edges we can skip till we have three nodes. The formula about the number of edges that can be skipped till there are N nodes selected using this form of construction is given under the explanation section.

EXPLANATION:

Let us first sort the edges in **non-decreasing** order. We intend to skip some edges by connecting them to the vertices which are already in the **MST** and try to increase the weight of the **MST** as much as possible. Start with a set of selected vertices S with a single vertex 1. Iterate over the edges from $i=1$ to M, every edge that is selected or included in the **MST** must add a node that is not included in S till then and every edge that is skipped must have an edge between two nodes of S at that time. Let at this moment the set S contains V vertices. Maximum number of edges in a set of V vertices is $\frac{V\cdot(V-1)}{2}$ out of which exactly $V-1$ are included in the **MST** till now. This means i^{th} edge must be included in the answer if number of edges skipped till this point is $=\frac{V\cdot(V-1)}{2}-(V-1)$. If in the end some nodes remain unconnected because of skipping we can connect them to vertex 1 directly using the largest weight edges which are not used.

TIME COMPLEXITY:

$O(Nlog(N))$ for each test case.

PREREQUISITES:

segment tree

PROBLEM:

You are given N segments where each segment is of the form $[L_i, R_i]$ $(1 \leq L_i \leq R_i \leq N)$.

It is ensured that for any two segments X and Y, one of the following conditions hold true:

- $X \cap Y = X$,
- $X \cap Y = Y$, or
- $X \cap Y = \emptyset$ (empty set)

Note that $\cap$ denotes the intersection of segments.

In other words, for any two segments, either one is completely inside the other or they are completely disjoint.

The i^{th} segment $(1 \leq i \leq N)$ is assigned a value V_i.
A subset $S = \{S_1, S_2, \ldots, S_k\}$ having k segments is considered *good* if there exists an integer array B of size k such that:

- All elements of the array B are **distinct**.
- B_i lies in the segment S_i.

Let the score of a *good* subset be defined as the sum of values of all segments of the subset. Find the **maximum** score of a *good* subset.

EXPLANATION:

First sort the segments given in the input in increasing order of their length (number of elements in them). Now iterate over the segments in the ascending order of their length. Now, Suppose we are at some segment $[L, R]$. There are two cases:

- There is a point in $[L, R]$ which has not been assigned to any interval yet. Then, we can freely assign it to this interval.
- Every point has been assigned to some segment. Then, find the lowest value point in the interval and check if it is more profitable to assign it to $[L, R]$.

Both the cases listed above can be combined into one by assuming that every point is initially assigned a value of 0. The calculation of the minimum and its position in a range along with point updates can be done efficiently using a segment tree.

For details of implementation please refer to the solutions attached.

TIME COMPLEXITY:

$O(Nlog(N))$ for each test case.

PREREQUISITES:

Maximum sum subarray

PROBLEM:

You have two arrays A and B. In one move, you can choose either the first or last element of B, and move it to either the front or back of A.

What is the maximum possible sum of a subarray in the final array?

EXPLANATION:

We'd like a large subarray sum, so it makes sense to ignore negative elements from B as much as possible and take as many positive elements as we can.

One obvious way to do this is to just move all the negative elements to one side and positive elements to the other — for example, move all negative elements of B to the back of A, and all positive elements of B to the front of A. Alternately, we can move all the negative elements to the front and the positive ones to the back.

It turns out that these are the only two cases we need to consider!

Proof

This can be proved by analyzing what the final answer looks like.

- If the maximum subarray doesn't contain any elements from B, it doesn't matter how the distribution is done, since subarrays of A remain unchanged.
- If the maximum subarray doesn't contain any elements from A, the best we can do is obviously just the sum of all positive elements in B, which both cases of ours cover.
- Finally, suppose the maximum subarray includes elements from both A and B. Note that since elements of B can only be inserted at the start or end, the structure of such a subarray is:
  - Some elements of B and a prefix of A, or
  - Some elements of B and a suffix of A

"Some elements of B" is optimally every positive integer of B, and then we choose a prefix or suffix of A with maximum sum. Note that this is exactly what our two cases cover, hence completing the proof.

There are only two cases. For each one, find the maximum subarray sum of the resulting array in linear time (this is a well-known algorithm, and is linked above in the prerequisites).

The final answer is the maximum of the two cases.

TIME COMPLEXITY

$O(N)$ per test case.

PROBLEM:

You are given two integers X and Y. Find three integers A, B, C in the range $[-1000, 1000]$ whose mean is X and median is Y.

EXPLANATION:

The median is Y, so one of the three numbers should definitely be Y.

The mean is X, which means that $\frac{A+B+C}{3} = X$, or in other words, $A + B + C = 3X$.

After noticing this, several constructions are possible.

One solution is to print the three numbers Y, Y, $3X - 2Y$. Their sum is clearly $3X$, and since there are two occurrences of Y, one of them is guaranteed to be the middle element regardless of the value of $3X - 2Y$, hence the median is Y.

TIME COMPLEXITY

$O(1)$ per test case.

PREREQUISITES:

Dynamic Programming, MEX

PROBLEM:

A function F is defined on an array A of length N as follows:

$F(A) = \sum_{i=1}^{N} MEX(A_1 + A_2 + \cdots + A_i)$

For example, if $A = [2, 0, 1]$, then $F(A) = MEX([2]) + MEX([2, 0]) + MEX([2, 0, 1]) = 0 + 1 + 3 = 4$.

You are given two integers N and K. For each $i = N, N + 1, \ldots, K$ in order, find the number of **permutations** P of integers $[0, 1, 2, \ldots, N - 1]$ such that $F(P) = i$. Since the answer can be very large, find it modulo $(10^9 + 7)$.

EXPLANATION:

Hint

Try to put the elements $0, 1, 2, \ldots, N - 1$ in order and see how the value of $F(A)$ changes for putting a new element in the permutation

Solution

Let's say Pre_j denotes the prefix of length j of the permutation P and Mex_j denotes the mex of Pre_j. Thus $F(A) = Mex_1 + Mex_2 + \cdots + Mex_N$.

Initially consider that the permutation P is empty. Hence $Mex_j = 0$ for each $1 \leq j \leq N$, as the mex of an empty array is 0. Thus $F(A) = 0$. Now we put the elements from 0 to $N - 1$ in order.

Say we put the element 0 on index id_0. Now for all indices $j = id_0, id_0 + 1, \ldots, N$, Pre_j will contain the element 0 making $Mex_j = 1$, for the rest indices Mex_j remains 0. So putting the element 0 on index id_0 increase the value of $F(A)$ by $N - id_0 + 1$.

Now we put the element 1 on index $id_1 (id_1 \neq id_0)$. Here two cases occur:

- $id_1 < id_0$: For all indices $j = id_0, id_0 + 1, \ldots, N$, Pre_j will contain the element $0, 1$ making $Mex_j = 2$. So putting the element 1 on index id_1 increases the value of $F(A)$ by $N - id_0 + 1$, because for each $id_0 \leq j \leq N$ the value of Mex_j becomes 2 from 1, Mex_j remains unchanged for the remaining indices.
- $id_1 > id_0$: For all indices $j = id_1, id_1 + 1, \ldots, N$, Pre_j will contain the element $0, 1$ making $Mex_j = 2$. So putting the element 1 on index id_1 increases the value of $F(A)$ by $N - id_1 + 1$ because for each $id_1 \leq j \leq N$ the value of Mex_j becomes 2 from 1, Mex_j remains unchanged for the remaining indices.

Let' say $Mx_1 = \max(id_0, id_1)$. Then putting 1 on index id_1 increases the value of $F(A)$ by $N - MX_1 + 1$.

If we put the element 2 on index id_2 and $Mx_2 = \max(id_0, id_1, id_2)$, it will increase the value of $F(A)$ by $N - MX_2 + 1$.

Let, $Dp[i][Mx_i][S]$ = number of ways to put the elements $0, 1, \ldots, i$ in a permutation of length N such that maximum position of these elements is Mx_i and the value of $F(A)$ we get is S. So $Dp[0][i][N - i + 1] = 1$ for each $1 \leq i \leq N$.

Say, we have put the elements $0, 1, 2, \ldots, i - 1$ on positions $id_0, id_1, \ldots, id_{i-1}$ respectively, the current value of $F(A)$ is S. Now we put the element i on index id_i and $Mx_i = \max(id_0, id_1, \ldots, id_i)$. It will increase the value

of $F(A)$ by $N - mx[i] + 1$. Here two cases may occur:

- $id_i = Mx_i$ (i.e . we put the element i on index Mx_i): The maximum position among $0, 1, \ldots, i-1$ (i.e. Mx_{i-1}) can lie in the range $[1, Mx_i - 1]$. Thus we will do: $Dp[i][Mx_i][S + N - Mx_i + 1] += \sum_{j=1}^{Mx_i - 1} Dp[i-1][j][S]$.
- $id_i < Mx_i$ (i.e one of $0, 1, \ldots, i-1$ on position Mx_i): We have put $0, 1, \ldots, i-1$ among the first Mx_i positions from left. Hence we can put the element i on one of the $Mx_i - i$ positions. Thus we will do: $Dp[i][Mx_i][S + N - Mx_i + 1] += Dp[i-1][Mx_i][S] * (Mx_i - i)$.

So the number of permutations P of length N with $F(A) = i$ will be $Dp[N-1][N][i]$.
.

TIME COMPLEXITY:

$O(N^2 \cdot K)$ for each test case.

PROBLEM:

You have two strings A and B. You'd like to make them equal.
You can either right-rotate A_i to reach B_i for positive cost, or right-rotate B_i to reach A_i, for negative cost.

Find the minimum **absolute** cost possible.

EXPLANATION:

Let's first convert every A_i to B_i using right rotations. At the i-th index, this has a cost of:

- $B_i - A_i$, if $A_i \leq B_i$
- $B_i - A_i + 26$, if $A_i > B_i$.

Let this cost for index i be C_i, and $S = \sum_{i=1}^{N} C_i$ be our initial cost.
Now, we want to use the $B_i \rightarrow A_i$ right rotation at some indices instead to make the absolute value of S as low as possible.
Let's see how this changes S:

- First, we must subtract C_i from S, since we aren't using the $A_i \rightarrow B_i$ rotation anymore.
- Then, we add the cost of the $B_i \rightarrow A_i$ rotation to S. Note that this is exactly $-(26 - C_i)$ (you can derive this from the definition of C_i given above).

So, our change is $S \rightarrow S - C_i - (26 - C_i) = S - 26$.

This means it doesn't matter which index we choose, the score is only going to decrease by 26.

So, the possible scores we can obtain are $\{S, S - 26, S - 2 \cdot 26, S - 3 \cdot 26, \ldots, S - N \cdot 26\}$. The answer is thus the minimum absolute value among all these $N + 1$ values, which can easily be found in $\mathrm{O}(N)$.

Note that there is an even simpler implementation: since we can only subtract 26, the value of S can always be brought into the range $[-26, 26]$.
The answer is thus simply the minimum of $(S \bmod 26)$ and $((-S) \bmod 26)$.
For example, if $S = 100$, the answer is the minimum of $(100 \bmod 26) = 22$ and $(-100 \bmod 26) = 4$, which is 4.

TIME COMPLEXITY

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

PREREQUISITES:

Depth-first Search, Dynamic Programing

PROBLEM:

You are a given a tree of N vertices. The i-th vertex has two things assigned to it — a value A_i and a range $[L_i, R_i]$. It is guaranteed that $L_i \leq A_i \leq R_i$.

You perform the following operation **exactly once**:

- Pick a (possibly empty) subset S of vertices of the tree. S must satisfy the additional condition that no two vertices in S are directly connected by an edge, i.e, it is an **independent set**.
- Then, for each $x \in S$, you can set A_x to any value in the range $[L_x, R_x]$ (endpoints included).

Your task is to minimize the value of $\sum |A_u - A_v|$ over all **unordered** pairs (u, v) such that u and v are connected by an edge.

EXPLANATION:

Observation : For a particular node u if we are going to change its value, we will either make it equal to the median of all A_i values of its neighbors or either it would be equal to L_u or R_u depending upon which is the nearest value to the median.

Median minimizes the sum of absolute deviations

Given a set of values S
We're basically after:
arg min(x) $\sum_{i=1}^{N} |si - x|$

One should notice that $d|x|/dx = sign(x)$
Hence, deriving the sum above yields $\sum_{i=1}^{N} sign|si - x|$
This equals to zero only when the number of positive items equals the number of negative which happens when $x = median[s_1, s_2, \cdots, s_N]$.

If median is less than L then we will set A_u to L: Suppose we set A_u to some value $L' > L$. Reducing L' by 1 does not increase the absolute sum of deviations(number of values $\geq L'$ is less than or equal to number of values $<$ than L') and thus we can keep on decreasing the value of L' until we reach L.
Similarly, if median is greater than R we will set it to R.

Now this problem can be solved using dynamic programing.
Let us root the tree at node 1. For each node of the tree calculate the optimal value after a change by sorting all the values of the neighbors and using the above observation and let this value for node i be stored in $optimal_i$.
Let $dp[u][1]$ represent the minimum value that can be obtained in the subtree of node u by selecting some or more nodes in set S while also changing the value of node u and dp[u][0] represents the minimum value that can be obtained in its subtree by selecting some or more nodes in set S while not changing the value of node u. If we change the value of node u we cannot change the value of any of its neighbors. Initialize all the dp states with 0.

start a depth first search from node 1, Now
$dp[u][1] = \sum dp[x][0] + abs(A[x] - optimal[u])$ for every child x of node u
$dp[u][0] = \sum min(dp[x][0] + abs(A[x] - A[u]), dp[x][1] + abs(optimal[x] - A[u]))$ for every child x of node u

The answer to the problem is $min(dp[1][0], dp[1][1])$

TIME COMPLEXITY:

$O(Nlog(N))$ for each test case.

PROBLEM:

Given two strings A and B, rearrange them so that the length of their longest common subsequence is minimized.

EXPLANATION:

Consider some character α. Let it occur $f_A(\alpha)$ times in A and $f_B(\alpha)$ times in B. Then, the length of $LCS(A, B)$ is at least $\min(f_A(\alpha), f_B(\alpha))$, no matter how they are rearranged.

Proof

This should be obvious: simply consider the subsequence consisting only of this character. It has length $\min(f_A(\alpha), f_B(\alpha))$, and is present no matter what the rearrangement is.

This means the answer is, at the very least, the maximum value of $\min(f_A(\alpha), f_B(\alpha))$ across all characters α from a to z.

It turns out that this is exactly the answer: if you sort A in ascending order and B in descending order, then there are no common subsequences with more than one distinct letter, so the answer is simply the longest common subsequence consisting of a single letter.

TIME COMPLEXITY

$O(N)$ per test case.

PROBLEM:

You have an array of length N. In one move, you can replace 2 elements by their gcd. What is the minimum possible sum of the final array?

EXPLANATION

Let $G = \gcd(A_1, A_2, \ldots, A_N)$. The answer is simply $N \cdot G$.

Clearly, it is not possible to make any element less than G because G divides every element of A. Conversely, it is possible to make every element equal to G - to make the i-th element equal to G, perform the operations on $(1, i), (2, i), (3, i), \ldots, (N, i)$.

All that remains is to compute G. This is easy - using the fact that $\gcd(a, b, c) = \gcd(a, \gcd(b, c))$, we can simply initialize G to A_1, and then run a loop of i from 2 to N, each time doing $G \leftarrow \gcd(G, A_i)$.

TIME COMPLEXITY:

$O(N + \log M)$ per test case, where M is the maximum element of the array.

PREREQUISITES:

Dynamic programming

PROBLEM:

You are given integers N and M.
An $N \times M$ tables containing 0 and 1 is said to be *special* if, for each (i,j), $A_{i,j} = 1$ if and only if the sum of row i equals the sum of column j.

Find the minimum sum of values of a special table with N rows and M columns.

EXPLANATION:

First off, what does a special table even look like?

Suppose the row sums are $r_1, r_2, \ldots, r_N$ and the column sums are $c_1, c_2, \ldots, c_M$.
Say $r_1 = k$. What information does that give you?

Answer

There are exactly k ones in the first row.
This means that there are exactly k columns whose sum is k.

First, let's assume $k \neq 0$.
Applying the same argument to one of the columns with sum k tells us that there are also exactly k rows whose sum is k.

So we have k rows and k columns, such that:

- The intersection of any one of these rows and columns contains a 1
- All other cells in these rows/columns are 0

This gives us k^2 ones in the grid.

Notice that without loss of generality, we can assume $r_1 = r_2 = \ldots = r_k = c_1 = c_2 = \ldots = c_k = k$, then delete these rows and columns and apply the same argument to the rest of the grid.

Now, what happens if $k = 0$?
Well, we still have exactly k columns with sum k, i.e, there are no columns with sum 0.
This means we can simply delete the first row and consider the remaining part of the grid instead.

The above discussion in fact tells us the following:

- First, the analysis of $k = 0$ tells us that the grid can contain *either* empty rows or empty columns, but not both. So, we can safely delete empty rows/columns and solve for the reduced N or M appropriately.
- Second, let the *distinct* non-zero row (or column) sums be $x_1 < x_2 < \ldots < x_d$.
  Then, the grid contains exactly $x_1^2 + x_2^2 + \ldots + x_d^2$ ones.
- Finally, notice that the number of non-zero rows will equal the number of non-zero columns. Combining this with the first point, it's enough to deal with only square grids, i.e, solving for an $N \times M$ grid is the same as solving for a $\min(N,M) \times \min(N,M)$ grid.

Now let's use the above observations to formulate a solution.
From now on, I'll assume $N = M$ since we established that that's the only case that matters.

As in the second point above, let $x_1, x_2, \ldots, x_d$ be the distinct row-sums of the grid.
Then, note that we must have $x_1 + x_2 + \ldots + x_d = N$.

So, our objective is to minimize $x_1^2 + x_2^2 + \ldots + x_d^2$ subject to the above constraint. Note that the x_i are all distinct here.

Since $N \leq 5000$, solving this in $\mathrm{O}(N^2)$ is still ok: and that is exactly what we will do using dynamic programming.

Let $dp(i,j)$ denote the minimum possible value of the sum of squares, such that:

- The sum of the elements chosen is i; and
- We have picked elements from $\{1, 2, \ldots, j\}$.

Under this formulation, the answer is simply $dp(N,N)$.
Computing $dp(i,j)$ is not hard: we either choose j or we don't, so

$$dp(i,j) = \min(dp(i,j-1), dp(i-j,j-1) + j^2)$$

This allows us to solve a single testcase in $\mathrm{O}(N^2)$.

However, note that there are a large number of testcases, and so running this $\mathrm{O}(N^2)$ algorithm separately for each is not quite enough.

However, the dp states themselves don't actually change across testcases: the values computed are the exact same.
So, simply precompute the dp table for all possible values before even processing any testcase: then, a testcase can be answered in $\mathrm{O}(1)$.

TIME COMPLEXITY

$\mathrm{O}(N^2)$ precomputation followed by $\mathrm{O}(1)$ per test case.

PROBLEM:

A normal year contains N days, while a “Mob” year contains $N + M$ days. Every K-th year is a “Mob” year.

Given X, does the $(X - 1)$-th day fall in a “Mob” year?

EXPLANATION:

Let’s look at a consecutive set of K years, starting from the first.
Years $1, 2, \ldots, K - 1$ are not “Mob” years, while year K is.

In terms of days, this means the first $(K - 1) \times N$ days aren’t in a “Mob” year, while days $(K - 1) \times N + 1$ to $K \times N + M$ are in one.
Notice that this already gives us the answer for when $1 \leq X \leq K \times N + M$.

What happens when $X > K \times N + M$?
Well, by symmetry we can just ignore the first $K \times N + M$ days!
That is, the answer for X is the same as the answer for $X - (K \times N + M)$.

So, we keep subtracting $K \times N + M$ from X till we reach a point where $1 \leq X \leq K \times N + M$, then use the condition we derived above to answer for this X.

However, this is too slow: for example, if $K = N = M = 1$, then we’d be subtracting 2 from X at each step, which is way too little when X is large.

Instead, notice that the operation we are performing is exactly the modulo operation!
That is, we are essentially computing $X \pmod{K \times N + M}$.

So, simply compute this quantity, for example using the % operator in most languages.
Notice that we want the result to be between 1 and $K \times N + M$, so if the result is zero then set it to $K \times N + M$.

Finally, use the condition derived above to answer the query. This gives us a solution in $O(1)$ per test case.

TIME COMPLEXITY

$O(1)$ per test case.

PROBLEM:

Stack likes number 3 a lot.

He has two non negative integers a and b.
He gets happy if atleast one of them to be divisible by 3.

In order to get happy, he can perform some moves(possibly 0).

In one move,

- he can change a to $abs(a-b)$, or
- he can change b to $abs(b-a)$

Find the minimum number of moves after which Stack gets happy.

It can be proved that Stack can get happy after performing finite number of moves.

EXPLANATION:

There are three cases:
Case $(a$ % $3 == 0||b$ % $3 == 0)$: The answer is 0.
Case $(a$ % $3 == b$ % 3 && b % $3! = 0)$: The answer is 1 as subtracting the minimum of a or b from their maximum results in a number divisible by 3.
Case $(a$ % $3! = b$ % 3 && b % $3! = 0$ && a % $3! = 0$) : Let a % $3 == 1$ and b % $3 == 2$(otherwise swap them), subtract a from b twice to get a number divisible by 3. The answer is 2 in this case.

TIME COMPLEXITY:

$O(1)$ for each test case.

PROBLEM:

There is a town with N people and initially, the i^{th} person has A_i coins. However, some people of the town decide to become *monks*. If the i^{th} person becomes a monk, then:

- He leaves the town thereby reducing the number of people in the town by 1.
- He distributes X $(0 \leq X \leq A_i)$ coins to the remaining people of the town (not necessarily equally). Note that each monk can freely choose his value of X, and different monks may choose different values of X.
- He takes the remaining $A_i - X$ coins with him.

For example, initially, if $A = [1, 3, 4, 5]$ and 4^{th} person decides to become a monk then he can leave the town and can give 2 coins to the 1^{st} person, 1 coin to the 2^{nd} person, no coins to the 3^{rd} person and take 2 coins along with him while going. Now A becomes $[3, 4, 4]$.

Determine the **minimum** number of people who have to become monks, so that in the end, everyone remaining in the town has an equal number of coins.

EXPLANATION:

Let B be the set of people who become monks and C be the set of people who do not become monks. For all the people in C to have an equal number of coins, it is sufficient and optimal to distribute coins from the people in B to the people in C so that all the people in C have as much coins as the maximum coins held initially by any person in C.

Let Max_C be the maximum coins held initially by any person in C, $Size_C$ be the size of C, Sum_B be the sum of the coins held initially by all the people in B, and Sum_C be the sum of the coins held initially by all people in C. It is clear that we have Sum_B coins to distribute among people in C and the minimum number of coins required so that all the people in C have Max_C coins is $Max_C \times Size_C - Sum_C$. So, this selection of B and C is possible if $Sum_B \geq Max_C \times Size_C - Sum_C$.

C is characterized by Max_C so we can go over all possible C by iterating over each $A[i] = Max_C$. This means that all elements of A having values higher that $A[i]$ will fall into B. This iteration is very easy if A is sorted because $Size_C$ can be found easily using the index i, Sum_C is the prefix sum till index i, Sum_B is suffix sum for index $i + 1$, and Max_C is $A[i]$.

We will pick the split (of B and C) which is valid (satisfies the condition) and has minimum $Size_B$ or maximum $Size_C$ as required in the question.

TIME COMPLEXITY:

$O(Nlog(N))$ for each test case.

PREREQUISITES

Number Theory (Properties of GCD), Two-pointers, Segment Tree

PROBLEM

Choose a subarray S of array A such that if we divide this subarray into 2 **non-empty** subsequences having GCDs GCD_1, GCD_2 respectively, then:

$MXDV(S) = max(GCD_1 - GCD_2) \geq K$ for **atleast one pair** of subsequences.

Find the **smallest** possible length of subarray S.

QUICK EXPLANATION

- $GCD_1 - GCD_2$ will have a maximum value when the **length of first subsequence is** 1 and the **second subsequence contains the rest of the elements** of the parent sequence.
- The element in the first subsequence can either be the **maximum** or the **second maximum** element of the subsequence.
- We can use **two-pointers** and maintain a set with pairs of {element, index} for finding the max and second max of the current subarray along with a **Segment Tree** for range GCD queries.

EXPLANATION

Observation 1

$GCD_1 - GCD_2$ will have a maximum value when the length of first subsequence is 1 and the second subsequence contains the rest of the elements of the parent sequence.

Proof

GCD of a sequence either decreases or remains same on adding additional elements to the sequence. Hence, you will always try to decrease the length of the first subsequence and increase the length of the second subsequence.

Observation 2

The optimal choice for the first subsequence would always be *one of the largest* 2 *distinct elements* of the array.

Proof

Let all the *distinct elements* in an array A in sorted order be:-

$A_1 < A_2 < \ldots < A_{N-1} < A_N$. (Here, A_{N-1} is not necessarily the second last element of the array, these are just distinct elements in sorted order)

Now if we take the largest element of this sequence as the first subsequence then,

$GCD_1 - GCD_2 = A_N - gcd(A_1, A_2, \ldots A_{N-1})$

and we also know that:

$A_{N-1} - A_{N-2} \geq gcd(A_{N-1}, A_{N-2})$, as $A_{N-1} - A_{N-2} > 0$ and GCD divides both elements A_{N-1}, A_{N-2}.

$\Rightarrow A_{N-1} - A_{N-2} \geq gcd(A_1, A_2, \ldots, A_{N-2}, A_{N-1})$, as GCD of a sequence decreases or remains same on increasing length of the sequence.

$\Rightarrow A_N - A_{N-2} \geq gcd(A_1, A_2, \ldots, A_{N-2}, A_{N-1})$ as $A_N > A_{N-1}$.

$\Rightarrow A_N - gcd(A_1, A_2, \ldots, A_{N-2}, A_{N-1}) \geq A_{N-2}$

$\Rightarrow A_N - gcd(A_1, A_2, \ldots, A_{N-2}, A_{N-1}) > A_{N-2} - gcd(A_1, \ldots, A_{N-3}, A_{N-1}, A_N)$, as we can always say $gcd(A_1, \ldots, A_{N-3}, A_{N-1}, A_N) \geq 1$

Hence, it will **always** be better to choose $[A_N]$ as our first subsequence instead of $[A_{N-2}]$. In the same way it can be proved for all the other elements less than or equal to A_{N-2}. So our optimal choice would always be to choose one of the largest 2 *distinct* elements of the array.

Note: Increasing frequency of elements does not affect their GCD. All the occurrences of the largest and the second largest element will give the same answer so we need to consider them only once and no element smaller than or equal to A_{N-2} is suitable as already mentioned.

Extra Observation

If the largest element occurs more than once, the first subsequence can only contain the largest element for the most optimal answer.

Proof

Let's say our sequence is:- $A_1 \leq A_2 \leq \ldots \leq A_X < A_{X+1} = A_{X+2} = \ldots = A_N$

If we take A_N in the first subsequence, the GCD of the second subsequence will be $G_i = gcd(A_1, A_2, \ldots, A_{N-1})$.

First, we remove A_X from the second subsequence. The GCD of the second subsequence can either **increase or remain the same**.

Now, we will add A_N to the second subsequence. The gcd will remain the same as $A_{N-1} = A_N$ is **already present** in the second subsequence. Let's call the gcd of the second subsequence *now* as G_f.

Then, we can surely say $G_i \leq G_f$ and we already know $A_X < A_N$. So, $A_N - G_i > A_X - G_f$. Hence, in this case we only need to consider the largest element of the subarray.

Corner Case

If all elements of the array are equal, then if $K = 0$, the answer is 2 and -1 otherwise. This solution covers this case automatically but some solutions might find this as a corner case.

Reduced problem

So, the problem is reduced to finding the smallest subarray such that $A_X - GCD($ rest of elements of the subarray $) \geq K$ where, A_X is either the largest or the second largest distinct element.

Now, we will use two pointers to find the smallest subarray satisfying above conditions.

Our goal is to calculate the value of $MXDV(S)$ for the current subarray $[l, r]$ where, l and r are the current indices of our two-pointers. For calculating $GCD_1 - GCD_2$, we simply calculate the value:

A_X (largest or second largest element) $-$ GCD of the rest of the elements in the subarray.

If we know the index of A_X then this value can be calculated using a GCD Segment tree.

Now, for finding the indices of the largest and the second largest element (A_X) in the subarray, we can **map** the elements of the subarray with a **queue** containing indices of all the occurrences of the elements in the subarray. We can then **pop** and **push** indices in this map as we move our **left** and **right** pointers respectively.

If we use the *Extra Observation* mentioned, then this can be implemented using **std::set** which stores the elements as pairs of {elements, index}.

Note: You can even use binary-search instead of two-pointers. It still passes all the test cases.

TIME COMPLEXITY

The time complexity is $O(N \cdot \log(N) \cdot \log(A_i))$.

PREREQUISITES:

Prefix sums

PROBLEM:

Chef has a binary string A of length N. He creates binary string B by concatenating M copies of A. Find the number of positions in B such that $pref_i = suf_{i+1}$.

EXPLANATION:

Let S denote the sum of A, i.e, $S = A_1 + A_2 + \ldots + A_N$.

Now, suppose we know that $pref_i = suf_{i+1}$ for some index i of B. What can we say about $pref_i$?

Answer

$pref_i$ must be equal to $\frac{M \cdot S}{2}$.

This is because $pref_i + suf_{i+1}$ always equals the total sum of B, which is $M \cdot S$ (since B is formed from M copies of A).

Note that the above division is *not* floor division. In particular, when $M \cdot S$ is odd, no good index can exist.

Now the problem reduces to finding the number of indices of B whose prefix sum is a given value. This can be done in several ways, though they all depend on the fact that the prefix sums are non-decreasing. For example:

- Since M is small, it is possible to simply iterate over the number of copies while the current prefix sum is smaller than the target value, each time adding S to the current prefix sum. When the prefix sum exceeds the target, iterate across that copy of A in $\mathrm{O}(N)$ and count the number of good indices. This takes $\mathrm{O}(N + M)$ time.
  - Some care needs to be taken when implementing this. For example, it might be that the next copy of A (if it exists) also contributes some indices to the answer, for example if A starts with a 0. Also, depending on implementation, a string with all zeros might be an edge case for the solution, causing either TLE or WA since all $N \cdot M$ indices are good.
- Another option with less thinking involved is to binary search for the first and last positions with the target prefix sum. The prefix sum for a given position can be calculated in $\mathrm{O}(1)$ if we know the prefix sums of A, and so this solution runs in $\mathrm{O}(\log(N \cdot M))$. It still requires $\mathrm{O}(N)$ time to read the input string, however.

TIME COMPLEXITY:

$\mathrm{O}(N + M)$ or $\mathrm{O}(N + \log(N \cdot M))$ per test case, depending on implementation.

PREREQUISITES

Combinatorics

PROBLEM

You are given three positive integers N, M and K.

An array of integers $A_1, A_2, \ldots, A_N$ is *good* if the following statements hold:

- $0 \leq A_i \leq M$ for each $(1 \leq i \leq N)$
- $(A_i \& K) \leq (A_{i+1} \& K)$ for each $(1 \leq i \leq N-1)$
- $(A_i \mid K) \leq (A_{i+1} \mid K)$ for each $(1 \leq i \leq N-1)$

Here, $\&$ denotes the **bitwise AND operation** and $\mid$ denotes the **bitwise OR operation**.

Count the number of good arrays $A_1, A_2, \ldots, A_N$. As the result can be very large, you should print it modulo $998\,244\,353$.

EXPLANATION

First increament M by 1 so we have to consider only $0 \leq A_i < M$.

Small Observation

Let us consider only first 21 bits.
Let P contains the set bits in K, while Q contains the unset bits of K.
Then the array should be non-decreasing with respect to the bits present in P as well as Q.

General Approach while solving this problem

As we want $A_i < M$ so we will choose some prefix of M and mismatch the prefix only at the last index so that the later bits can be dealt independently.
For example if the binary representation of M is 110110, then we will consider the prefixes $1, 11, 1101, 11011$ and now changing the last bit we get $0, 10, 1100, 11010$.
Note that we have left the prefixes $110, 110110$ as we cannot make them smaller by changing the last bit.
Now after fixing the prefix S of length x we will ensure that the prefix of length x of every A_i should be $\leq S$ as well as prefix of A_N (which is the largest element of the array) should be exactly S (so that we don't do any overcounting later).
Now after fixing some prefix S and maintaining the above inequalities we have established that all the A_i will be $< M$ so we can independently fill the remaining bits of the suffix.

Now after fixing some prefix S we need to count the number of good Arrays A such that

- prefix of $A_i \leq S$
- prefix of $A_N = S$
- A is non-decreasing with respect to the bits of sets P and Q defined earlier.

After finding these counts for every prefix we can directly add them to obtain the answer. There will be no overcounting as we have fixed the prefix of A_N everytime.

Now let us consider $M = 1011100111$ in binary. Here Red color denotes the bits involved in set P, while Green color denotes the bits involved in set Q.

Let us consider the prefix of length 5 which is 10111. Now changing the last bit of that prefix we get $S = 10110$.
Let us compress the bits of both colors in the prefix now. So for Red bits in the prefix we need to ensure that Red

bits of every A_i should be ≤ 11 (i.e. 3 in decimal). For Green bits in the prefix we need to ensure that Green bits of every A_i should be ≤ 010 (i.e. 2 in decimal). As there are no inequality restrictions with respect to M in the suffix so we just want that the Red bits value in suffix of A should be $\leq 2^3 - 1$ (Since there are 3 red bits in the suffix) and Green bits value in suffix of A should be $\leq 2^2 - 1$ (Since there are 2 green bits in the suffix).

Let's say that Red bits value and green bits value of the prefix of A_N be x and y respectively (In the taken example $x = 3$ and $y = 2$.
Let's say that Red bits value and green bits value of the suffix of A_N be p and q respectively (In the taken example $0 \leq p \leq 2^2 - 1$ and $0 \leq q \leq 2^3 - 1$).
We will first update x to $2^3 \cdot x$ (Since there are 3 red bits in the suffix) and update y to $2^2 \cdot y$ (Since there are 2 green bits in the suffix).
Now total Red value $= x + p$ and total green value $= y + q$.
We want Red value as well as Green value of A to be non-decreasing.
Let us fix the fix the Red Value of A_N to $x + p$ and Green Value of A_N to $y + q$, then the number of non-decreasing arrays satisfying the above conditions is:

$$\left(\binom{n+x+p}{n} - \binom{n+x+p-1}{n} \right) \cdot \left(\binom{n+y+q}{n} - \binom{n+y+q-1}{n} \right)$$

This formula can be derived using stars and bars. In the above formula the first term denotes the contribution of Red bits while second term denotes the contribution of Green bits. Also the term $\binom{n+x+p}{n}$ denotes the number of non-decreasing arrays of length n such that all the elements are $\leq x + p$ and ≥ 0. So subtracting $\binom{n+x+p-1}{n}$ we will get the count of non-decreasing arrays whose last term is $x + p$.

So for each prefix we will add the above formula to obtain the answer. See editorialists code for the above implementation.

Some common Mistakes

Keep your $MaxN = 4 \cdot 10^6$ for precompution of factorials.

TIME COMPLEXITY

$O(log^2 m)$ for every test case

PREREQUISITES:

Bitwise operations

PROBLEM:

You are given N and X. Find the number of integers K such that $0 \leq K < N$ and $((N \oplus K)\ \&\ X) = 0$.

EXPLANATION

Since this task deals with bitwise operations, it makes sense to look at things bit by bit.

If the binary representation of a number y has a 1 in the i-th position, we say the i-th bit is *set* in y.

Now, we want the bitwise and of $N \oplus K$ with X to be 0. This means that if a bit is set in X, it will be set in K if and only if it is set in N. In other words, if some bit of X is set, there is exactly one choice for such a bit in K.

After this, we apply a classical technique when counting things that must be lexicographically smaller than something else. Iterate over the positions, and fix the one that is the first to be **strictly less**, then we have complete freedom in all following positions.

That is, iterate bits from 30 down to 0. Suppose we are currently at the i-th bit. We will assume that the bits from 30 to $(i+1)$ of both N and K are equal, and the i-th bit of K is strictly less than the i-th bit of N (of course, note that this is only possible when the i-th bit of N is 1, and the i-th bit of X is not 1).

Now, bits 0 to $i-1$ are totally free (except for the restriction on set bits of X mentioned before), and each of these have 2 choices (can be 0 or 1). So, if there are c such free bits, we add 2^c to the answer.

The final algorithm looks like this:

- Iterate i from 30 down to 0.
- If the i-th bit is either set in X or not set in N, do nothing
- Otherwise, count the number of bits from 0 to $i-1$ that are not set in X. If this number is c, add 2^c to the answer.

This can be implemented in $\mathrm{O}(\log N)$ by maintaining the running sum of bits that are not set in X; however a more naive implementation in $\mathrm{O}(\log^2 N)$ will still be fast enough.

TIME COMPLEXITY:

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

PROBLEM:

A string S is called *good* if none of the substrings of S of length ≥ 2 are palindromic.

For e.g. $S = \text{hello}$ is not good but $S = \text{world}$ is good.

You are given a string A of length N (containing lowercase Latin letters only). Rearrange A to form a good string or determine it is not possible to do so.

Recall that a string is called palindromic if it reads the same backwards and forwards. For example, noon and level are palindromic strings.

EXPLANATION:

Observation: If in a string there are no palindromic sub-strings of length 2 or 3, then there are no palindromic sub-strings in the string(except for sub-strings of length 1).

Keeping this observation in mind, we divide the entire string into three containers c_1, c_2, c_3 where, $c_i = \{j,\ such\ that\ j\ mod\ 3 = i, 1 \leq j \leq n\}$. In order to avoid palindromic sub-strings of length 3, we must keep a distance of at least 2 between each same character or we must keep them in same containers. Thus for each character we have a limit of its frequency as:

$$limit = \lceil \frac{n}{3} \rceil$$

If frequency of any character exceeds this limit then it won't be possible to make it a good string.

Now let's talk about the case when there are multiple characters that have frequency equal to limit. Taking another variable $afford$ as $(n\ mod\ 3)$.

- $afford = 0$: All containers are of length $\lceil \frac{n}{3} \rceil$.
- $afford = 1$: Only one container is of length $\lceil \frac{n}{3} \rceil$.
- $afford = 2$: Two containers are of length $\lceil \frac{n}{3} \rceil$.

Let us define bad as number of characters having frequency equal to $limit$. If bad is more than the number of containers having length equal to $\lceil \frac{n}{3} \rceil$, then it won't be possible to construct a good string.

If none of such cases arise then we can confidently say that we can create a good string.

Now we loop through each characters in descending order of their frequency and if its frequency is equal to $limit$, then we fill it in all the positions of container whose size is equal to that of $limit$ else we can fill its characters in the containers in the order $c_3 -> c_2 -> c_1$

TIME COMPLEXITY:

$O(Nlog(N))$, for each test case.

PREREQUISITES:

Prefix sums

PROBLEM:

An array is said to be *good* if no subarray has a xor of 0.

Given an array A, in one move you can replace one of its elements with any non-negative integer. Find the minimum number of moves to make A good.

EXPLANATION:

Let $P_i = A_1 \oplus A_2 \oplus \ldots \oplus A_i$ denote the prefix xor of array A, with $P_0 = 0$.

Note that a subarray $[L, R]$ can have a xor of zero if and only if $P_R \oplus P_{L-1} = 0$, i.e, $P_R = P_{L-1}$.
So, an array A is *good* if and only if all its prefix sums are distinct.

Now look at what our given operation does to the prefix sums: changing the element A_i changes exactly all the prefix sums $P_i, P_{i+1}, \ldots, P_N$.
In particular, suppose we set $A_i \leftarrow x$. Let $y = A_i \oplus x$. Then, each P_j for $j \geq i$ becomes $(P_j \oplus y)$.

This allows us to 'fix' the array from left to right, as follows:

- Let S be the set of current prefix sums. Initially, $S = \{0\}$.
- Iterate i from 1 to N.
  - If P_i is not in S, insert it to S and continue.
  - if P_i is in S, we have no choice but to perform an operation. We might as well perform this operation on position i. By choosing a large enough value of x and setting $A_i \leftarrow x$, we can ensure the following:
    - P_i is no longer in S
    - For any $j \geq i$ and $k < i$, it is impossible for $P_j = P_k$ to ever happen.
  - In other words, we can essentially just pretend we are starting from an entirely new array. The current set S is useless to us, so we can simply clear it.
  - Note that S should now contain something denoting the 'empty' prefix (recall that we initially had $S = \{0\}$ for this purpose). There are a couple of ways of achieving this:
    - If the values of P_i were calculated in advance, insert P_{i-1} into S (the editorialist's code does this).
    - Otherwise, notice that the values of P_i can actually be calculated on-the-go. If this is how you to choose to implement it, simply reset the current prefix sum to 0 and insert 0 into S to simulate starting from a new array (the setter's code does this).

TIME COMPLEXITY

$\mathrm{O}(N)$ or $\mathrm{O}(N \log N)$ per test case, depending on implementation.

PROBLEM:

Given N, count the number of pairs (A, B) such that $1 \leq A, B \leq N$ and $A + B$ is odd.

EXPLANATION:

For $A + B$ to be odd, one of A must be odd and the other must be even. In fact, this is the only restriction.

So, to count the total number of pairs:

- Let the number of even numbers from 1 to N be E, and the number of odd numbers from 1 to N be O.
- If A is odd and B is even, we have $O \cdot E$ choices for them (O for A and E for B)
- If A is even and B is odd, we have $E \cdot O$ choices for them.
- So, the total number of choices is $2 \cdot E \cdot O$.

All that remains is to find E and O quickly. This is quite easy:

- $O = \lceil \frac{N}{2} \rceil$, which can be implemented in most languages as `(N+1)/2`, using integer division.
- $E = \lfloor \frac{N}{2} \rfloor$, which can be implemented in most languages as simply `N/2` using integer division

Note that $N \leq 10^9$, so the answer can exceed the range of 32-bit integers. Make sure to use a 64-bit integer datatype.

TIME COMPLEXITY

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

PROBLEM:

Kulyash stays in room that has a **single bulb** and N buttons. The bulb is initially **on**. The initial states of the buttons are stored in a binary string S of length N, where S_i is the i the button. If Kulyash toggles any single button then the state of the bulb reverses i.e. the bulb lights up if it was off and vice versa. Kulyash has toggled some buttons and the final states of the buttons are stored in another binary string R of length N . We need to find the final stage of the bulb.

EXPLANATION:

The important things note before we beigin are:

- The initial state of the bulb is On.
- Even If a button is toggled the state of bulb reverses. Also, the state of bulb reverses for every single toggle.
- Initial state of buttons are defined in string S.
- Final state of buttons are defined in string R.

-If a single button is toggled the final state of the bulb will be 0, as the initial state is 1.
-If two buttons are toggled then the state of bulb reverses twice and stays as 1.

-Thus we can conclude that if number of buttons toggled is an even number then the state of bulb stays as 1 else if its an odd number, its state stays as 0.

TIME COMPLEXITY:

Time complexity is $O(n)$.

PROBLEM:

For two positive integers a and b, let $g(a, b) =$ **gcd** $(a, b)+$ **lcm** (a, b).

For a positive integer N, let $f(N)$ denote the **minimum** value of $g(a, b)$ over all the pairs of positive integers (a, b) such that $a + b = N$.

Find out the number of **ordered** pairs (a, b) such that $a + b = N$ and $g(a, b) = f(N)$.

EXPLANATION:

Observation 1: For all $1 \leq a \leq N$, which are factors of N,

$$gcd(N - a, a) = a$$

Proof:
By the property

$$gcd(A, B) = gcd(A - B, B)$$

We can write

$$gcd(N - a, a) = gcd(N - 2a, a) = gcd(N - (\frac{N}{a} - 1)a) = gcd(a, a) = a$$

Also using the property

$$gcd(a, b) \times lcm(a, b) = a \times b$$

we get,

$$g(a, N - a) = a + \frac{a \times (N - a)}{a} = N$$

which is the minimum value we can get. To prove this let us assume a is not a factor of N.
Then $g(a, N - a) = gcd(a, N - a) + \frac{a(N-a)}{gcd(a,N-a)}$.
Let us assume:

$$L = N - g(a, N - a)$$
$$L = N - gcd(a, N - a) - \frac{a(N - a)}{gcd(a, N - a)}$$

Simplifying it we get

$$L = (1 - \frac{a}{gcd(a, N - a)})((N - a) - gcd(a, N - a))$$

Since $gcd(a, b) \leq min(a, b)$, therefore we can see that the first term of L will always be ≤ 0 while the second term will always be ≥ 0, so we conclude that

$$L \leq 0 => N \leq g(a, N-a)$$

Thus we can select all possible factors of N as a and b would be $(N-a)$. This way we can find all possible pairs (a, b) that gives minimum value of $g(a, b)$.

TIME COMPLEXITY:

$O(\sqrt{N})$, for each test case.

PREREQUISITES:

Bitwise OR operation

PROBLEM:

Chef has a sequence A of N integers. He picked a **non-negative** integer X and obtained another sequence B of N integers, defined as $B_i = A_i \mid X$ for each $1 \leq i \leq N$. Unfortunately, he forgot X, and the sequence B got jumbled.

Chef wonders what the value of X is. He gives you the sequence A and the jumbled sequence B. Your task is to find whether there is a non-negative integer X such that there exists a **permutation** C of B satisfying the condition $C_i = A_i \mid X$ for each $1 \leq i \leq N$.

If there are multiple such X, find **any** of them.

Note: Here $\mid$ represents the **bitwise OR** operation.

EXPLANATION:

Let us look at each bit one at a time. If the i^{th} bit is set to 1 in a elements of the array A and set to 1 in b elements of the array B. Then these two conditions are necessary for an answer to exist:

- $b \geq a$

Why?

$(0|1 = 1)$
Suppose the i^{th} bit is set to 1 in the binary representation of X, then it will be set to 1 in all numbers in array B or else it will be set to 1 in exactly same number of elements of array B as that of array A in which it was set to 1.
$\implies b \geq a$

- $(a < b) \implies b = N$

Why?

$(0|0 = 0)$
Suppose the i^{th} bit is set to 0 in the binary representation of X, then it will be set to 1 in exactly same number of elements in array B as that of array A in which it was set to 1. Therefore $a < b$ means i^{th} bit is set to 1 in X which implies that it has to be set to 1 in all values of array B as $(0|1 = 1)$ as well as $(1|1 = 1)$.
$\implies (a < b) \rightarrow b = N$

But these are necessary conditions not sufficient.

Example

Let $A = [1, 2]$ and $B = [0, 3]$ necessary conditions are true but an answer does not exist.

Maintain the count of each bit in array A and array B.Initialize $X = 0$. Calculate the value of X by iterating on the bits and verifying the necessary conditions. If count of i^{th} bit in $B = N$ add 2^i to X.(This value X is same as taking $bitwise$ & of all values of array B)
Lastly check whether the calculated X is correct by generating Array B' where B'_i = $A_i|X$.Now if array B is same as B' then X is an answer otherwise output -1.

TIME COMPLEXITY:

$O(Nlog(N))$ or $O(Nlog(N) + Nlog(max(A_i)))$ for each test case

PROBLEM:

There are three hidden numbers A, B, and C. You know the values of $A \vee B$, $B \vee C$, and $A \vee C$. How many possible tuples (A, B, C) can satify this?

EXPLANATION:

The OR operation is bitwise independent, so we can simply find the number of possibilities for each bit from 0 to 19 and multiply them all together to get the final answer.

Now, let's look at a specific bit. For this bit, each of $A \vee B, B \vee C, A \vee C$ is either 0 or 1.

There are a few cases here:

- If all 3 are zero, then this bit must be zero in all of A, B, C so there is only one option.
- If exactly one of them is 1, such a case can never happen so the answer is immediately zero (for example, if $A \vee B = 1$, then either A or B must be 1 at this bit, and so at least one of $A \vee C, B \vee C$ must be 1).
- If exactly two of them are 1, there is exactly one option (if $A \vee B = 0$, then both A and B must be 0 and so $C = 1$ is the only option, and so on).
- If all three are 1, there are 4 possible options:
  - One way is for all three of A, B, C to have a 1 at this bit
  - Otherwise, we can also choose exactly two of them to have a 1 at this bit. There are 3 ways to choose a pair.
  - Adding up the above options, we have 4 valid possibilities.

Compute the appropriate multiplier for each bit, then multiply them all together to obtain the final answer.
Note that the answer can be as large as 4^{20}, so make sure you use a 64-bit integer datatype.

Alternately, once a bit is fixed, one can also simply iterate across all $2^3 = 8$ possibilities of values of A, B, C and see how many of them contribute to the current configuration.

TIME COMPLEXITY

$O(\log N)$ per test case, where $N = 2^{20}$.

PREREQUISITES:

Bitwise operations

PROBLEM:

Chef has an array A of length N such that $A_i = i$.

In one operation, Chef can pick any two elements of the array, delete them from A, and append either their **bitwise XOR** or their **bitwise OR** to A.

Note that after each operation, the length of the array decreases by 1.

Let F be the final number obtained after $N - 1$ operations are made. You are given an integer X, determine if you can get $F = X$ via some sequence of operations.

In case it is possible to get $F = X$, print the operations too (see the section Output format for more details), otherwise print -1.

EXPLANATION:

Let us handle the case when $N = 2$ separately, when N is 2 only possible value of X is 3 as both XOR and OR of 1 and 2 result in 3. So if N is 2 and $x! = 3$, answer is -1 otherwise both XOR or OR of 1 and 2 are acceptable answers.
If some bit in X is set to 1 which is not set to 1 in any number till N then there is no possible way to attain such X , Hence answer in this case is -1 too.
If N is a power of two then the most significant bit in N is set to 1 in only one number and is 0 in all other numbers less than N hence whether you take XOR with this number or OR with this number, this bit has to be set to 1 in X . If it is unset in X then the answer is -1

Otherwise it is always possible to construct the answer:
Consider the construction take OR of all the elements in the array which are not a power of 2.
This way at least all the bits upto most significant bit of N(excluding it) will be set in the the number obtained.

Proof

If N is a power of 2.Then since we take OR with $(2^{msb}) - 1$. It sets all the bits upto msb of N in the number obtained.
Otherwise we take OR of N with $2^{msb} - 1$ and this sets all the bits upto msb of N in the number obtained.

Now iterate on the bits from $i = 0$ to the most significant bit of N. If it is set in X we take OR with 2^i present in the array otherwise we take XOR with 2^i present in the array.
(Since if N is a power of 2 then X will have most significant bit of N set, therefore we will take OR with N in the end.)
Thus we can get the final value same as X by this method.

TIME COMPLEXITY:

$O(N)$ or $O(Nlog(N))$ depending upon implementation for each test case.

PREREQUISITES:

Counting inversions in a permutation

PROBLEM:

You are given N strings, all of length M. In one move, you swap swap the i-th character of two adjacent strings. Find the minimum number of moves to make every string a palindrome.

EXPLANATION:

For a string S of length M to be a palindrome, we must have $S_i = S_{M+1-i}$ for each $1 \leq i \leq M$.
In particular, for N strings to all be palindromes, the following must hold:

- Let C_i denote the string formed by the i-th characters of all the strings. That is, $C_i = S_{1,i}S_{2,i} \ldots S_{N,i}$. C_i is essentially the string formed by the i-th 'column' of the strings.
- Then, it must hold that $C_i = C_{M+1-i}$ for every $1 \leq i \leq M$.

Note that the given operation only allows us to move a character within its own column — it cannot be moved to a different column.

So, we only really need to compute the following:

- For each $1 \leq i \leq M$, compute the minimum number of adjacent swaps required to make $C_i = C_{M+1-i}$, when swaps can be performed on both strings.
- Then, add this answer up for all pairs of columns.
- If any pair of columns cannot be made equal, the answer is -1.

Thus, we are left with a subproblem: given two strings S and T, find the minimum number of moves to make $S = T$, given that you can make adjacent swaps on either of them.

This can be solved with the help of an interesting observation: it is enough to only perform the swaps on one string, and not change the other!

Proof

Suppose we make swaps on both S and T, and they are equal in the end.
Let the last swap on T be $(i, i+1)$.

Then, we can instead not perform this swap on T, and perform it as the last operation on S.
This still keeps $S = T$, while performing one less operation on T.

Repeatedly applying this will bring us to a state where $S = T$ but no operations have been made on T, as required.

So, let's keep T constant, and find the minimum number of swaps on S needed to make $S = T$.
This is a rather well-known problem, and you can find a tutorial **here** for example.

The basic idea is as follows:

- For each position i, compute the position where S_i should end up. Denote this by pos_i.
- This can be done somewhat easily as follows:
  - Let's look at each character individually.
  - The first occurrence of 'a' in S should end up as the first occurrence of 'a' in T, the second occurrence in S should end up as the second occurrence in T, and so on.
  - This applies to each character.
  - Finding this can be done by simply knowing the positions of the characters in S and T.

- Once we know the positions where each character ends up, the problem essentially asks for the minimum number of swaps to reach this configuration.
- This is simply the number of inversions in the pos array, which can be computed in $\mathrm{O}(N \log N)$ in a variety of ways.

A related problem you can try is **1430E**.

TIME COMPLEXITY

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

PREREQUISITES:

Dynamic programming

PROBLEM:

Chef has N numbers written on his board. He wants only one number to remain on the board and so he devises a special tactic to do so.

He randomly picks up 2 numbers written on the board say A and B respectively. He erases both of them from the board and writes a new number on the board which equals -

- $A - B$ if parity of $A =$ parity of B
- $A + B$ if parity of $A \neq$ parity of B

Here parity refers to remainder obtained when a number is divided by 2.

Notice that the total numbers on the board reduced by 1 after 1 such operation. After a finite number of operations there would only be 1 number remaining on the board.

Chef is interested in knowing how many different final numbers he could end up with.

EXPLANATION:

This problem requires a little bit of case work and observations. In the end the resulting number is a signed(positive or negative) sum of all numbers of the array but there are some constraints on the sign of the numbers.

Observation 1

Let x be the count of odd numbers in the array then $floor(x/2)$ odd numbers will have a '-' sign before them. After applying operations some numbers will have positive sign in result and some negative let us show no matter how you apply operations you always end up with always $floor(n/2)$ negatives (here n = count of odds) (say this function as F_n)
For $n = 1$ => negative = 0
For $n = 2$ => negative = 1
Now when computing for higher n we split it considering before the last operation one side had processed x_1 numbers and other x_2 such that
$x_1 + x_2 = n$ then result after this operation will be F_{x_1} + $(x_2 - F_{x_2})$ iff both x_1 and x_2 have same parity else it will be $F_{x_1} + F_{x_2}$
For $n = 3$ => only splitting possible $(2, 1)$ or $(1, 2)$ => $(2, 1) = 1 + 0 = 1$ and $(1, 2) = 0 + 1 = 1$; Hence, 1;
for $n = 4$ => $(2, 2)(3, 1)(1, 3)$ => $(2, 2) = 1 + (2 - 1) = 2$;$(3, 1)$ => $1 + (1 - 0) = 2$; $(1, 3) = 0 + (3 - 1) = 2$;
Now lets generalize by induction $(F_n = n/2)$
For odd n we will always get different parity numbers hence F_{x_1} + F_{x_2} = $x_1/2 + x_2/2 = (x_1 + x_2)/2 = n/2$
For even n we will always get same parity numbers hence $F_{x_1} + (x_2 - F_{x_2})$
now for odd numbers $floor(n/2) = (n - 1)/2$
same parity - (even, even) - $x_1/2 + (x_2 - x_2/2)$ for even numbers $x_2 - x_2/2 = x_2/2$ hence $(x_1 + x_2)/2 = n/2$
(odd, odd) - $x_1/2 + (x_2 - x_2/2)$ lets substitute $(x_1 - 1)/2 + (x_2 - (x_2 - 1)/2) = (x_1 + x_2)/2 = n/2$
Hence, we always end up with $floor(n/2)$ odd negatives no matter how we procced.

Observation 2

If the count of odd numbers is greater than or equal to 2 then the sign of any even number can be assigned to be + or -.
Construction: Select an odd number, add all even numbers to it in which you want a + sign. Select another odd

number , add all even numbers to it in which you want a - sign. Both of these resulting numbers are odd. Now subtract the second number from the first number. The resulting number is the only even number and rest all are odd. Keep on doing the operations until there is only one number left. The result number either has the configuration of '+' and '-' as we need or the reverse of it(+ instead of - and vice versa). If we have achieved reverse configuration just swap all even numbers in the beginning to get the desired number.

Observation 3

If there is No odd number in the array we cannot assign - sign or + sign to all the even numbers. Just look at the first operation the sign of the elements in the first operation is swapped and will always be opposite to each other no matter what the operations are. These means there is at least 1 even number with + sign and one with - sign.
Construction: Subtract all even numbers in front of which you want a + sign except one from the even number(in front of which you want a negative sign). Now subtract the even number formed by these operations from the one which was excluded (in front of which we need a + sign). Now all the even elements in front of which we want a + sign have a + sign, Keep on subtracting even numbers from this number till you reach a single number.

Observation 4

If the count of odd numbers in the array is exactly 1 then we cannot assign '-' sign to all even numbers. If there is only one even number this already true. Let there be more than 2 even numbers. The count of odd numbers is always going to be 1 in the array and '-' sign can only be introduced when two even numbers are subtracted from each other in this case. Now these two even numbers cannot have the same sign till the end. To create a number in which all even numbers have positive sign just keep adding all of them to the odd number.
Construction: Add all ,except one, even numbers in front of which you want a positive sign to the odd number and keep subtracting other even numbers in front of which you want a negative sign from the even number excluded earlier. Now add these two numbers to get the desired number.

After these operations this problem reduces to compute all possible sums given these restrictions on the sign of even and odd numbers in various cases. This can be solved using standard dynamic programing approach.
For details of implementation please refer to the solutions attached.

TIME COMPLEXITY:

$O(N^3 \cdot A_{MAX})$ for each test case.

PROBLEM:

Given a binary string of length $2N$, partition it into non-palindromic substrings or report that this is impossible.

EXPLANATION:

A trivial impossible case is when S consists of only 0's or only 1's.

In every other case, a valid partition exists: in fact, you only need 2 substrings at most.

How?

Let's consider a few cases.

Suppose S is itself not a palindrome. Then, no more partitioning needs to be done: just take S itself.

Now we consider the case when S is a palindrome. Note that the length of S is even.

- Suppose the first half of S is not a palindrome. Then, its second half is also not a palindrome, so simply partition it into these two halves
- Otherwise, the first N characters of S form a palindrome. In this case, consider the string formed by the first $N + 1$ characters of S: this is definitely not a palindrome.

Proof

Suppose it was a palindrome. Note that the first N characters of this string are also a palindrome. Together, this information tells us that all the characters of this substring must be the same.

However, if this were to be the case, then S being a palindrome would make all the characters of S the same, which is a contradiction since we took care of that case at the very beginning.

The exact same argument shows that the first (and hence last) $N - 1$ characters of S also don't form a palindrome. So, we can split S into its first $N + 1$ and last $N - 1$ characters to obtain a solution.

TIME COMPLEXITY

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

PROBLEM:

You are given an integer N. Let A be an $N \times N$ grid such that $A_{i,j} = i + N \cdot (j-1)$ for $1 \leq i,j \leq N$. For example, if $N = 4$ the grid looks like:

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

You start at the top left corner of the grid, i.e, cell $(1, 1)$. You would like to reach the bottom-right corner, cell (N, N). To do so, whenever you are at cell (i,j), you can move to either cell $(i+1,j)$ or cell $(i,j+1)$ provided that the corresponding cell lies within the grid (more informally, you can make one step down or one step right).

The *score* of a path you take to reach (N, N) is the sum of all the numbers on that path.

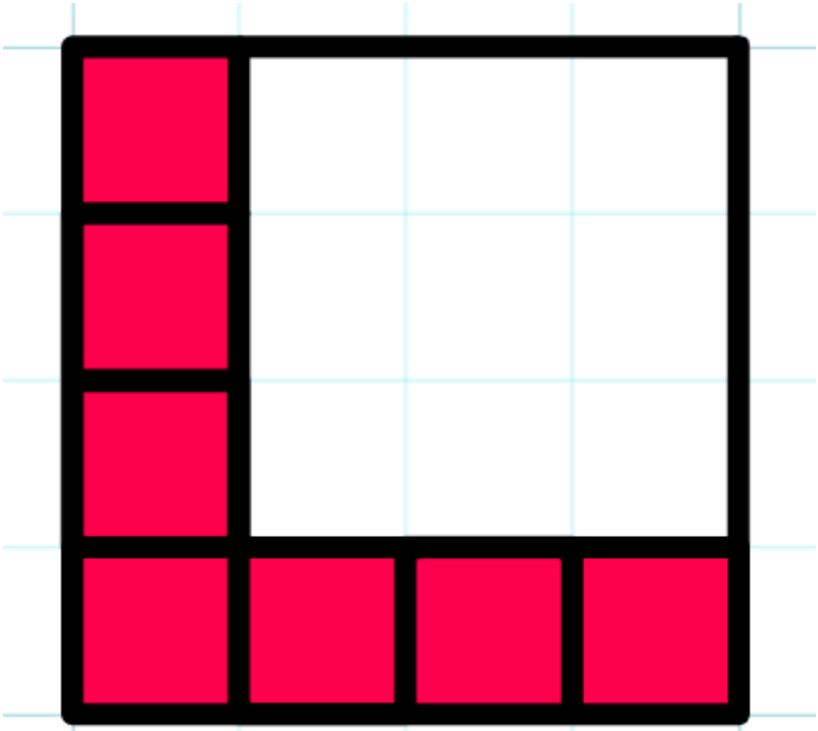
You are given an integer K that is either 0 or 1. Is there a path reaching (N, N) such that the parity of its score is K?

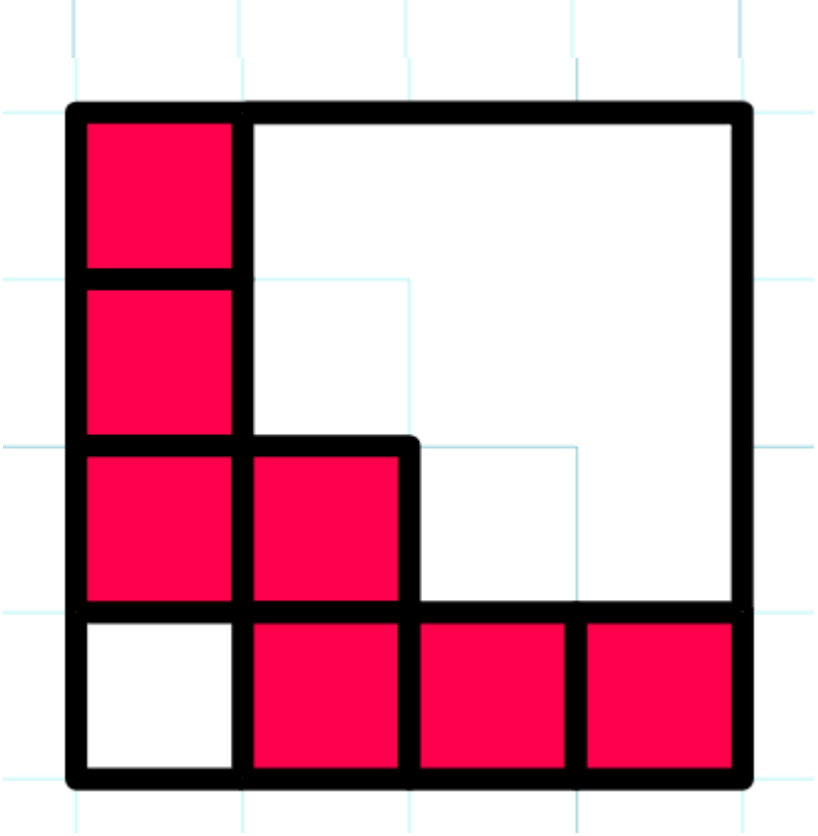
Recall that the parity of an integer is the (non-negative) remainder obtained when dividing it by 2. For example, the parity of 246 is 0 and the parity of 11 is 1. In other words, an even number has parity 0 and an odd number has parity 1.

EXPLANATION:

When N is even, an answer always exists irrespective of the value of K.

N is even





Choose a path similar to one of the two paths shown above. We can show that these paths have different parities and exactly one of these paths has parity equal to K. There is a difference of one cell in both of these paths and these cells have different parity.
$A_{N,1} = N \cdot (N-1) + 1$ which is odd as N is even
$A_{N-1,2} = N \cdot (N-2) + 2$ which is even as N is even
Hence both of these paths have different parities and we can choose one which has the parity of sum equal to K

In case when N is odd, answer exists only when $K = 1$

N is odd

All paths of odd length (number of cells here) end at an odd value and all the paths with even length end at at even value. For length 1 it is true as $A_{1,1} = 1$ odd.
Let us suppose this is true for general length L. Let the path end at an index $A_{i,j}$ and the length of path is odd.

This means by our assumption $A_{i,j}$ is odd.
$\therefore A_{i,j} = N \cdot (i-1) + j$ is odd
$\implies A_{i,j+1} = N \cdot (i-1) + j + 1$ is even
$\implies A_{i+1,j} = N \cdot (i) + j = A_{i,j} + N$ is even (as N is odd)
Similarly we can show the same for even length paths.
Therefore the claim is true by induction. Since, N is odd therefore the path length is odd and the paths looks like $odd -> even -> odd -> \ldots . \ldots odd$, where odd number occurs N times and even number occurs $N-1$ times. Odd(N) times addition of odd numbers is an odd number and addition of an odd number and even number is also odd. Therefore the parity of path in case of odd N is always odd.

TIME COMPLEXITY:

$O(1)$ for each test case.

PREREQUISITES:

Bitwise Operations

PROBLEM:

Chef's last task in his internship requires him to construct 2 **permutations**, each having length N, such that the **bitwise AND** of each pair of elements at every index is the same.

Formally, if the permutations are A and B, then, the value $A_i\&B_i$ must be **constant** over all $1 \leq i \leq N$.

Help Chef by providing him 2 such permutations. If it is not possible to construct such permutations, print -1 instead.

EXPLANATION:

In case $N = 1$, $A = [1]$ and $B = [1]$.
Now let's discuss for $N \geq 2$.

Observation 1: The only constant value possible is 0.
Any number greater than 1, say c would not be possible since then no number can be paired with 1 to give their AND as c. If we select the constant as 1, then also no number can be paired 2 to get their AND as 1. Thus by elimination of choice, only 0 is a viable candidate.

Observation 2: It is not possible if N is odd(and greater than 1).
Say for $N = 2k + 1$. There would be $k + 1$ odd numbers and k even numbers. Since each odd number has their first bit set and to unset it we would need an even number but since the number of odd numbers is more than that of even numbers, therefore it won't be possible to make such pairs.

Observation 3: if there are two numbers say, a and b, then if
sum of a and b is of the form $2^x - 1$, then their AND is 0.
Thus when N is even, we can find all such possible pairs by going from N to 1, and for each number i(not already included in a pair), we can find its pair k by subtracting i from p where p is formed by setting all unset bits(right of MSB) of i.

This way after finding all the pairs, we can construct our two permutations.

TIME COMPLEXITY:

$O(N)$, for each test case.

PROBLEM:

Alice has a permutation of $\{1, 2, \ldots, N\}$. She dislikes K of these numbers $B_1, \ldots, B_K$. Can you print the permutation with these numbers removed?

EXPLANATION:

This is an implementation problem more than anything else — it is enough to do exactly what is asked for.

Iterate across the values of A. Say we are currently at A_i, and we want to know whether to print it or not.
All that we really need to know is whether A_i is one of the elements of B, and to check this faster than $\mathrm{O}(N)$ (since the constraints are such that $\mathrm{O}(N^2)$ algorithms will TLE).

This check can be done in $\mathrm{O}(\log N)$ or even $\mathrm{O}(1)$, in several ways:

- The easiest way is to use the `set` data structure present in most languages. Insert all the elements of B into a set S, then simply check if A_i is in S. This check is $\mathrm{O}(\log N)$ in C++ `set` and Java `TreeSet`, and $\mathrm{O}(1)$ in python `set` and Java `HashSet`.
- Alternately, we can use an array. Note that all the elements are from 1 to N, so we can create an array $mark$ of length N, such that $mark_i$ is 1 if i is present in B and 0 otherwise. Now, we only need to look at $mark_{A_i}$ to decide if A_i is in B, which takes $\mathrm{O}(1)$ time.

TIME COMPLEXITY

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

PROBLEM:

Find a permutation of $[1, N]$, so that the sum of prefix-GCDs is exactly X.

EXPLANATION:

GCD of positive numbers is always ≥ 1. So the sum of N of the prefix-GCDs is going to be $\geq N$. So, if $X < N$, there is no answer, and hence output -1.

For all other cases, we can construct a permutation. We see from the input constraints that X is guaranteed to be $\leq 2 * N - 1$. So, we can place a suitable element at the first, and then a 1, and the rest in any order. In such a case, the GCD of all prefixes except the very first is 1. So, the first element has to be $X - (N - 1)$. So the permutation is $[X - (N - 1), 1, 2, 3, \ldots, N]$, taking care that the first element isn't repeated twice.

TIME COMPLEXITY:

Time complexity is $O(N)$.

PROBLEM:

Chef and Chefina are playing a game involving N piles where the i^{th} pile $(1 \leq i \leq N)$ has A_i stones initially.

They take alternate turns with Chef starting the game.

In his/her turn a player can choose any **non-empty** pile (say i^{th} pile) and remove X stones from it iff:

- **Parity** of X is **same** as parity of i.
- $1 \leq X \leq A_i$.

The player who cannot make a move loses. Determine the winner of the game if both players play optimally.

EXPLANATION:

This problem is based on
Sprague-Grundy theorem

Here we first calculate the grundy values of the piles:
$Case1$: Odd Piles:
Here we would note that if the pile has odd number of stones then grundy value of the pile would be 1 and if the number of stones is even then grundy value would be 0.

$Case2$: Even Piles:
Here the grundy value would be half the number of stones, that is for a pile having n stones, its grundy value would be $\lfloor \frac{n}{2} \rfloor$

Thus we would calculate the grundy values of all the piles and check the xor-sum of the grundy values.
If the xor-sum is 0, then $Chefina$ would win otherwise $Chef$ would win.

TIME COMPLEXITY:

$O(N)$, for each test case.

PREREQUISITES:

Observation, basic combinatorics

PROBLEM:

JJ has an array A of length N, where each element lies between 0 and $2^K - 1$. In one move, he can pick two integers x and i such that $0 \leq x < 2^K$ and $1 \leq i \leq N$, and:

- Set $A_j := A_j \ \& \ x$ for the prefix ending at i, or
- Set $A_j := A_j \mid x$ for the suffix starting at i

How many different arrays can he create by performing this operation several times?

EXPLANATION:

First, note that the problem can be solved for each bit independently since there is no limit on the number of operations. So, if we are able to solve the problem for $K = 1$, we can apply this solution to every bit separately and multiply all the answers together to obtain the final answer.

Now, we just need to solve the problem for $K = 1$, i.e, the array contains only 0-s and 1-s. When this is the case, our operations reduce to:

- Setting some prefix of the array to 0 (the first operation, with $x = 0$)
- Setting some suffix of the array to 1 (the second operation, with $x = 1$)

Note that performing the first operation with $x = 1$ or the second operation with $x = 0$ don't change the array at all, so we can safely ignore them.

This gives us the following observations:

- It is enough to perform at most one operation of each type
- The prefix chosen for the first operation and the suffix chosen for the second can be chosen in such a way that they don't intersect.
- If an operation is performed on the prefix of length i, and $A_i = 0$ initially, we can achieve the same result by performing the operation on the prefix of length $i - 1$ instead. So, it is enough to consider the prefix operation only for those i such that $A_i = 1$.
- Similarly, it is enough to consider the suffix operation only for those i such that $A_i = 0$.

Putting all this together, we can see that the problem simply reduces to counting the number of 10-subsequences of A (do you see why?), which can be done easily in $\mathrm{O}(N)$.

Don't forget to account for the case where no prefix operation and/or no suffix operation is performed!

As mentioned at the start, the final answer is obtained by solving this subproblem for each of the K bits, and multiplying all the answers together.

TIME COMPLEXITY:

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

PROBLEM

For 3 **distinct prime** integers A, B, and C $(1 < A, B, C < 2^{30})$, we define positive integers X, Y, and Z as:

$X = A \oplus B$, $Y = B \oplus C$, and $Z = C \oplus A$, where $\oplus$ denotes the **bitwise XOR** operation.

Given only **two** integers X and Y and the fact that **at least one** integer amongst X, Y, and Z is **odd**, find the values of A, B, and C.

EXPLANATION

First Observation:

If we have X and Y then we can get Z by taking bitwise XOR of the X and Y.

Proof

$X = A \oplus B$, $Y = B \oplus C$ and $X \oplus Y = (A \oplus B) \oplus (B \oplus C) = A \oplus C = Z$.

Second Observation:

It is given that at least one integer amongst X, Y, and Z is odd, this implies one of the numbers from A, B, and C must be even as bitwise XOR of even and odd number gives odd.

The only even prime number that exists is 2, hence we have concluded one number out of A, B, and C.

To obtain the remaining two numbers from A, B, and C, we can take bitwise XOR of 2 and two odd numbers out of X, Y, and Z.

Hence, we arrive at the values of all A, B, and C.

TIME COMPLEXITY

The time complexity is $O(1)$ per testcase.

PREREQUISITES:

Familiarity with prime factorization

PROBLEM:

Given two integers A and B, does every prime dividing B also divide A?

EXPLANATION:

The obvious solution is to attempt to prime-factorize B and then check if each prime factor divides A. Unfortunately, both the number of test cases and the numbers themselves are too large: even a fast prime-factorization algorithm likely isn't going to be fast enough, so we need to be a bit smarter.

There's a couple of different ways to approach this task, though for the most part they rely on looking at the prime factorizations of A and B and drawing some conclusions from that.

Method 1 (GCD)

Consider a prime p that divides B. Suppose it also divides A.
Then, p definitely divides $\gcd(A, B)$.
In particular, since $\gcd(A, B)$ is a factor of B, the answer to the initial question is "Yes" if and only if B and $\gcd(A, B)$ have the same set of primes dividing them.

So, we essentially reduce our problem to this: given two integers x and y, where y is a factor of x, do they have the same set of prime factors dividing them?

- If $x = 1$, then $y = 1$ and the answer is obviously "Yes".
- If $x > 1$ but $y = 1$, the answer is obviously "No".
- Otherwise, we have $x, y > 1$. Since $y \mid x$, let's look at $z = x/y$.
- Consider some prime p that divides x.
  - If x and y have the same power of p, then z doesn't contain p at all.
  - Otherwise, z does contain some power of p. So, p divides y if and only if p divides $\gcd(z, y)$.

In other words, in the case when $x, y > 1$, they have the same prime factors if and only if z and $\gcd(z, y)$ have the same prime factors!

This gives us a pretty simply algorithm:

- Initially, $x = B$ and $y = \gcd(A, B)$.
- While $x > 1$ and $y > 1$, replace x and y with x/y and $\gcd(y, x/y)$.
- In the end, if $x = 1$ the answer is "Yes" and otherwise the answer is "No".

The while loop runs only $O(\log B)$ times, since at each step the larger number is at least halved.
This is fast enough for our purposes.

Method 2 (Exponentiation)

A rather interesting solution is to simply compute $A^{60} \bmod B$ and check whether this is 0 or not. Computing $A^{60} \bmod B$ can be done quickly with exponentiation, however if you're using C++ or Java you'll need to use 128-bit integers or something like **this**, since otherwise the multiplications will overflow even `long long`.

Why does this work?

The idea behind this solution is simple: if p is a prime that divides B, then p divides A if and only if p divides A^{60}.
This follows from the fact that the largest power of a prime that fits within 10^{18} is 59 (since $2^{60} > 10^{18}$).

The prime factorization of A^{60} has every prime to a power of at least 60, which is strictly larger than the power of anything in B. So, every prime in B divides A^{60} if and only if then B itself divides A^{60}, so we simply check that instead.

TIME COMPLEXITY

$O(\log B)$ per test case.

CODE:

Setter's code (C++)
Tester's code (C++)
Editorialist's code (Python)

Quote

notmotivated Nov '22

Method 1 implementation: **CodeChef: Practical coding for everyone**

ketan_gupta Nov '22

Very nice question

ketan_gupta Nov '22

Really liked the second method…
Here is my `BigInteger` solution **Java BigInteger AC**

bakru_k78 Nov '22

This is new thing that I learn today
`largest power of a prime that fits within 10^18 ( 2^60 ) is 59` and new method of using python pow function `pow(x,y,x) = x^y % z`

theairo1 Nov '22

I stumbled upon the exact same problem. 😕

CHAPD Problem - CodeChef

Practice your programming skills with this problem on data structure and algorithms.

PREREQUISITES:

Algebraic manipulation

PROBLEM:

Given an integer N whose largest odd factor is at most 10^5, find two integers A and B such that $A^2 + B^2 = N$ or claim that none exist.

EXPLANATION:

There are several different constructions that can work in this task, so if you have an interesting one feel free to share it in the comments below.

The constraint on the largest odd factor is a bit weird, so let's try to use that. Note that it immediately implies that any large N must be even.

So, if we were able to obtain a solution for N from a solution for $N/2$, we could potentially use that to build a solution.

It turns out that the relationship is a bit stronger: when N is even, there exists an integer pair (A_1, B_1) such that $A_1^2 + B_1^2 = N$ *if and only if* there exists an integer pair (A_2, B_2) such that $A_2^2 + B_2^2 = N/2$.

Proof

Suppose $A^2 + B^2 = N/2$. Then, $(A + B)^2 + (A - B)^2 = 2A^2 + 2B^2 = N$ gives us a solution for N.

Conversely, if $A^2 + B^2 = N$, turning the above construction around gives us

$$(\frac{A+B}{2})^2 + (\frac{A-B}{2})^2 = \frac{A^2}{2} + \frac{B^2}{2} = \frac{N}{2}$$

The interesting thing here is that $(A + B)/2$ and $(A - B)/2$ are both integers: if N is even, the only way $A^2 + B^2 = N$ can have integer solutions is if both A and B have the same parity.

This tells us that it is enough to solve the problem for small N: we can reduce N to its largest odd factor, solve for this factor, then reconstruct the result for the original N using the method above.

Solving for $N \leq 10^5$ can be done with bruteforce: fix a value of A, then check if $N - A^2$ is itself a square integer. We only need to check those A such that $A^2 \leq N$, giving us a $\sqrt{10^5}$ solution, which is good enough.

There are also other constructions, though most use the same idea. One simple one is as follows:
Suppose we have a solution to $A^2 + B^2 = N$. Then, simply multiplying A and B by 2 gives us a solution to $4N$. So, we can simply divide N by 4 as long as possible, which will end with it being $\leq 2 \cdot 10^5$. Use the bruteforce to solve for this, then reconstruct the answer for the original N by multiplying by 2 as many times as needed.

PROBLEM:

You are given the position of a king on an 8×8 chessboard. Place the minimum number of queens such that they don't attack the king, but the king also cannot make a move.

EXPLANATION:

There are a couple of ways to solve this problem, though for the most part they all rely on the following observation:

- It is always possible to create a valid configuration using ≤ 2 queens

One way of approaching this problem is with casework. There are a handful of different cases that need to be covered, all needing one or two queens in different configurations:

- Case 1: the king is in one of the four corners (needs one queen)
- Case 2: the king is along the border of the board, but not in a corner (2 queens)
- Case 3: the king is one square away from the border and close to a corner (2 queens)
- Case 4: any position that is not in cases 1 and 2 (2 queens)

However, I will describe a simpler way to approach the problem, with far less thinking and casework required: simply bruteforce the positions of the queens!

It's obvious that 4 queens will always suffice to achieve what we want - just use one to cut off each row/column around the king.

After trying out a couple of cases, you might notice that you can always do it with 2 queens. However, maybe you aren't satisfied: perhaps there's a case you missed that needs 3 queens.

So, for now let's assume that we always need ≤ 3 queens, and bruteforce all options. The number of ways of placing ≤ 3 queens on the board is bounded by $64 \times 64 \times 64$. There are only 64 possible inputs, and for each combination of (king position, queen positions) we need to check whether each queen attacks the king and the squares around it, leading to a total of somewhere around $64^4 \cdot 9 \cdot 3$ operations.

At first glance, this is too many operations to perform within one second. However, the constant factor can be optimized in several ways:

- Note that we cannot place a queen at a position that attacks the king. Once the king's position is known, this removes every square from its row, column, and diagonals from consideration. This leaves us with somewhere between 30 and 40 valid positions, bringing down the constant to $64 \cdot 40^3$ instead (in practice, it's a lot less since all the middle positions give us ~30 valid squares.
- The above optimization is already enough to receive AC (in C++), however more optimizations can be made. For example, symmetry of the board allows us to solve for only the 8 cells of the form (i, j) where $1 \leq i \leq j \leq 4$, and then suitably reflect/rotate the answer to obtain the solution for any other cell
- Another option is to precompute all the answers offline, then store them in the code and simply print the answer when submitting. This essentially turns the time limit into 3 hours (the contest's duration) instead of one second.

Of course, noting that the answer is ≤ 2 also immediately makes even our first bruteforce fast enough since it cuts out a factor of 64.

PROBLEM:

You have an array A. In one move you can choose two integers $i < j$ such that $A_i = A_j$ and set $A_k = A_i$ for each $i < k < j$.
Can you perform operations such that A has at most two distinct elements in the end?

EXPLANATION:

First, if $A_1 = A_N$, the answer is clearly “Yes”: just make the entire array equal.

Otherwise, notice that the operation cannot change A_1 or A_N, so our only hope is to have the two distinct elements in the array be these two.
So, the only reasonable moves to make are those starting at $i = 1$ or those ending at $j = N$.

This immediately means the answer is “Yes” if and only if there exists an index i such that $A_i = A_1$ and $A_{i+1} = A_N$.

This can be checked easily with a simple `for` loop.

TIME COMPLEXITY

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

PROBLEM:

*CodeCheffers are aware that after a contest, all problems are moved into the platform's **practice section** . Based on user submissions during the contest, the system calculates and assigns a difficulty rating to each problem. Ideally, it is recommended that users practice problems in increasing order of difficulty.*

Our Chef has some students in his coding class who are practicing problems. Given the difficulty of the problems that the students have solved in order, help the Chef identify if they are solving them in non-decreasing order of difficulty. That is, the students should not solve a problem with difficulty d_1, and then later a problem with difficulty d_2, where $d_1 > d_2$.

Output "Yes" if the problems are attempted in non-decreasing order of difficulty rating and "No" if not.

EXPLANATION:

Attempting the problem in non-decreasing order implies that there does not exist an index i from 1 to $N-1$ such that $A_{i+1} < A_i$. If there exists such an index the answer is `NO` else the answer is `YES`.

TIME COMPLEXITY:

$O(N)$ or for each test case.

PREREQUISITES:

Knapsack-style dynamic programming

PROBLEM:

You have two arrays R and B, both of length N. At each index, you can choose either R_i or B_i. Let X denote the sum of all chosen R_i and Y denote the sum of all chosen B_i. Maximize $\min(X, Y)$.

EXPLANATION:

The limits on N and the values are small, so a natural knapsack-style dynamic programming solution should strike you, something along the following lines:

Let $f(i, x, y)$ be a boolean function, where $f(i, x, y)$ is true if and only if you can make choices among the first i elements such that the sum of reds is exactly x and the sum of blues is exactly y.
Transitions are extremely easy: $f(i, x, y) = f(i-1, x-R_i, y)$ V $f(i-1, x, y-B_i)$ (V denotes logical OR), and memoization naturally makes transitions $\mathrm{O}(1)$.
The final answer is the maximum value of $\min(x, y)$ across all (x, y) such that $f(N, x, y)$ is true.

While this is correct, it is also too slow. x and y can be as large as $200 \times N$, so we have $200^2 \times N^3$ states in our dp, which is way too much.

Note that the constraints do allow a solution in $\mathrm{O}(200 \times N^2)$, i.e, kicking out one state of our dp.
We can achieve that by a relatively common trick: turn the removed state into the value of the dp!

Consider a function $f(i, x)$ which denotes the maximum sum of blues from the first i elements, given that the sum of reds is x.
Transitions for this function are as follows:

- If we choose R_i, the sum of blues is $f(i-1, x-R_i)$
- Otherwise, the sum of blues is $f(i-1, x) + B_i$
- So, $f(i, x) = \max(f(i-1, x) + B_i, f(i-1, x-R_i))$

Once again, by memoizing $f(i, x)$ values, transitions are $\mathrm{O}(1)$, so both our time and space complexity are fine.

The final answer is the maximum of $\min(x, f(N, x))$ across all $0 \le x \le 200 \cdot N$.

TIME COMPLEXITY

$\mathrm{O}(N \cdot S)$ per test case, where $S = 200 \times N$.

PREREQUISITES:

Combinatorics

PROBLEM:

You have an $N \times M$ grid, each cell of which can be colored either red or green. The score of a colored grid is the number of right-down paths from $(1, 1)$ to (N, M) such that the number of red cells equals the number of green cells on this path.

Find the sum of scores across possible colorings of the grid.

EXPLANATION:

This task can be solved with a trick that's rather common in counting problems, sometimes called the *contribution trick*.

Whenever you need to fix a configuration and count the number of objects in it that satisfy some property, you can often instead fix an object that satisfies that property and count how many configurations it appears in — the sum of both values is the same.

Let's do the same thing here. We want to

- Fix a coloring of the grid
- Look at a right-down path from $(1, 1)$ to (N, M)
- Count it if it has an equal number of red and green cells

Instead, we will

- Fix a right-down path from $(1, 1)$ to (N, M)
- Color it in such a way that the number of red and green cells is equal
- Then, count the number of grids that contain this path

Summing this across all paths is the the final answer.

Now, let's see how to compute this.

- First, we need to fix a path. There are $\binom{N+M-2}{M-1}$ paths from $(1, 1)$ to (N, M) — this is a well-known result.
  - An easy proof is that you make exactly $N + M - 2$ steps in total with $M - 1$ of them being right and $N - 1$ being down, so fixing the positions of the right steps fixes the path.
- Next, we need to color the path correctly.
  - Let the length of the path be $L. Note that $L = N + M - 1$.
  - If L is odd, there is no way to color the path correctly
  - Otherwise, we simply fix the positions of the green cells in $\binom{L}{L/2}$ ways, which then also fixes the red cells
- Finally, we want to know how many grids this path appears in. We've fixed the colors of cells on the path, but everything else is completely free and has two choices each (red or green).
  - This gives us $2^{N \cdot M - L}$ grids that contain this path.

So, the final answer is simply

$$\binom{N + M - 2}{M - 1}\binom{L}{L/2}2^{N \cdot M - L}$$

where $L = N + M - 1$. Note that the answer is 0 when L is odd.

The binomial coefficients can be precomputed in a 2000×2000 table using Pascal's identity since we only need something like $\binom{2000}{1000}$ at worst. Similarly, the powers of 2 upto 10^6 can be precomputed.

This allows each query to be answered in $O(1)$ time.

There are of course other ways to compute binomial coefficients (for example, the standard factorial + inverse factorial method) and powers of 2 (via binary exponentiation), if you want to do something else.

PREREQUISITES:

Map data structure, Two pointers approach

PROBLEM:

You are given an array A of N integers. You must perform some (possibly zero) operations to make the elements of A distinct.

In one operation, you can either:

- Remove one element from the beginning of the array A and append any positive integer to the end.
- Or remove one element from the end of the array A and prepend any positive integer to the beginning.

Find the minimum number of operations required to make all the elements of the array distinct.

EXPLANATION:

It is trivial that after some operations the final array contains a prefix of distinct elements inserted into A followed by a subarray of A which contains distinct elements **or** a subarray of A which contains distinct elements is followed by a suffix of distinct elements inserted into A. For each index i where $1 \leq i \leq N$, we assume the subarray present in the final answer starts at this index and to decrease the number of operations we need the maximum length of such subarray for each index. For each index i where $1 \leq i \leq N$, we need to find maximum index j where $i \leq j$ such that the subarray $A[i,j]$ contains distinct elements. This can be done using two pointers approach in $O(Nlog(N))$.

Observation

Let the optimal answer contain the subarray $A[i,j]$,Let R denote the number of elements in the suffix of the array A starting from $j+1$ i.e $A[j+1,N]$ and L denote the number of elements in the prefix of array A ending at $i-1$. Let $L <= R$, then the answer cannot be less than $2 \cdot L + R$. This is because until all elements till index $i-1$ are deleted by performing operation of type 1, performing an operation of type 2 only increases the total number of operations. Thus it is optimal to have all similar operations together. Similarly the case when $R <= L$ can be analyzed.

The minimum number of operations required to delete all numbers upto index $i-1$ and from $j+1$ to N is $min(2 \cdot (i-1) + N - j, i - 1 + 2 \cdot (N-j))$. Minimum of this value over all indices i is the answer.

TIME COMPLEXITY:

$O(Nlog(N))$ for each test case.

PROBLEM:

Chef has an array A of length N.

In one operation, Chef can remove **any one** element from the array.

Determine the **minimum** number of operations required to make all the elements **same**.

EXPLANATION:

In order to make all the elements same, we will select the element with the maximum frequency and delete all the other elements. Thus we will calculate the frequency of each element and select the element say x with maximum frequency, say f. Then our answer would be:

$$n - f$$

where n is the size of the array.

In case there are more than element having same maximum frequency then we can select any one of them and our answer would still be the same.

TIME COMPLEXITY:

$O(N)$, for each test case.

PROBLEM:

Alice and Bob play a game on a circle with black and white pieces. On their move, a player can choose any subarray of this circle and left-rotate it once. As an extra condition, a move is only valid if the number of positions i with $A_i \neq A_{i+1}$ **strictly increases** after the move is made.

The person who cannot make a move loses. Determine who wins under optimal play.

EXPLANATION:

The main observation to be made here is as follows:

Let $diff(A)$ denote the number of adjacent pairs of indices with different values in A. Then, *any* valid move increases $diff(A)$ by exactly 2 - as such, a move is valid if and only if it increases $diff(A)$ by 2.

Proof

This is not too hard to see.

Suppose we perform the rotate operation on the range $[L, R]$. This is essentially the same thing as deleting A_L from its position, and then inserting it between A_R and A_{R+1}.

Let's analyze the deletion step:

- If $A_{L-1} = A_{L+1}$, deletion decreases $diff(A)$ by either 0 or 2, depending on the value of A_L
- If $A_{L-1} \neq A_{L+1}$, deletion doesn't change $diff(A)$ at all, no matter what the value of A_L is.

So, deleting A_L either reduces $diff(A)$ by 2, or leaves it the same.
The exact same analysis can be applied to the insertion step to see that insertion either increases $diff(A)$ by 2 or doesn't change it.

Thus, the only way to increase $diff(A)$ is for the deletion step to not change it, and the insertion step to increase it by 2. Ergo, the only possible increase is $+2$.

With this observation, the solution is fairly simple:

- Let x be the initial value of $diff(A)$.
  - x can be easily computed in $O(N)$.
- Let y be the maximum possible value of $diff(A)$
  - y can also be computed in $O(N)$. For $diff(A)$ to be maximum, we would like 0's and 1's to alternate as much as possible. Upon constructing a string like this, it can be seen that $y = 2 \cdot \min(c_0, c_1)$, where c_i is the count of i in A. Alternately, you can construct this alternating string and directly compute its difference value in $O(N)$.
- The number of moves that can be made is then always exactly $\frac{y-x}{2}$.
- If this value is odd, Alice wins. Otherwise, Bob wins.

TIME COMPLEXITY

$O(N)$ per test case.

PROBLEM:

You have a string S, K strings $P_1, \ldots, P_K$, and some characters given in a frequency array C.

For each x from 0 to K, find out whether it's possible to make S greater than exactly x of the given strings.

EXPLANATION:

One immediate observation we can make is that appending characters to S will never make it any smaller lexicographically: it can only make it larger.

Let's try to solve for a specific value of x first.

To make things easier for us, first sort the strings P_i in increasing order, i.e, $P_1 \leq P_2 \leq \ldots \leq P_K$.
Now, notice that making S greater than exactly x strings means we want to append some characters to S to make it satisfy $P_x < S \leq P_{x+1}$.
In particular, it's enough to only look at the two strings P_x and P_{x+1}, so let's do that.

From here, we can do a bit of basic casework to throw out a few simple cases, each time making what S we have to deal with more specific.

- If $S > P_{x+1}$, obviously the answer is `No`.
  - Now we have $S \leq P_{x+1}$.
- If $P_x = P_{x+1}$, again the answer is `No`.
  - Now we have $S < P_{x+1}$.
- If $S > P_x$, we don't need to append anything: we're already done and the answer is `Yes`.
  - Now we have $S \leq P_x < P_{x+1}$.
- If the length of S is greater than the length of P_x, then the answer is `No`.
  - Now we further have $|S| \leq |P_x|$.
- If S is not a prefix of P_x, once again the answer is `No`.
  - Now S is a prefix of P_x.

This is the end of the 'simple' cases, which don't depend on the extra characters in C at all.
From here on, we have to check whether appending some characters to S can make it $> P_x$.

This check can be done greedily position-by-position, starting from $i = N + 1$.
When we are at position i,

- If S is a prefix of both P_x and P_{x+1}, and we can append some character that is $> P_{x,i}$ but $\leq P_{x+1,i}$, we are immediately done and the answer is `Yes`.
- If S is not a prefix of P_{x+1}, then simply check if we can append *any* character $> P_{x,i}$: if we can, the answer is `Yes`.
- If the above cases are not possible, note that our only hope of making $S > P_x$ in the future is to append exactly $P_{x,i}$ and maintain S as a prefix of P_x, so check if this is possible.
  - If it is possible, do so. Make sure to update whether S is still a prefix of P_{x+1} or not.
  - If it is not possible, the answer is immediately `No`.
- A little bit of care needs to be taken when i is larger than the length of P_x and/or P_{x+1}: these need to be special-cased.

Note that this algorithm runs in something like $O(26 \cdot (|P_x| + |P_{x+1}|))$ time.

So, simply running this algorithm for every x is already fast enough! Every string is processed twice this way, giving us a solution in $O(26 \sum |P_i|)$ which is good enough.

Depending on your implementation, you might have to take special care of $x = 0$ and $x = K$, since the strings P_0 and P_{K+1} don't exist.

TIME COMPLEXITY

$O(N + K + 26 \sum |P_i|)$ per test case.

PREREQUISITES:

Sorting, sweep line and/or Dijkstra's algorithm

PROBLEM:

There are k points on the 2-D plane. The i-th point has a cost of c_i and energy e_i. You start at $(0, 0)$ and want to reach (N, M).

Moving one step up or right reduces energy by 1, while moving left or down increases it by 1. At the i-th of the the N points above, you can also choose to pay c_i and reset your energy to e_i.

Find the minimum cost to reach (N, M) while ensuring that your energy never becomes negative.

EXPLANATION:

Let us define the *height* of the point (x, y) to be $x + y$.
Note that moving from a point with height h_1 to a point with height h_2 will change your energy by exactly $h_2 - h_1$. In particular, if $h_2 < h_1$ it is always possible to make this movement without your energy falling negative.

A simple interpretation of the statement is to treat the points and costs as a weighted graph: create a graph with $k + 1$ vertices (where the first k are the given points and the $k + 1$-th is the destination), and edges as follows:

- Let h_i denote the height of the i-th point, as defined above.
- For $1 \leq u, v \leq k + 1$, if $h_v \leq h_u + e_u$, create an edge from u to v with weight c_u. Essentially, this says that we can move directly from u to v without having to reset energy at a different point.

The answer is now the shortest path from $(0, 0)$ to (N, M) in this graph.

However, this graph can have $\Omega(k^2)$ edges, which is too many. We need to optimize the above solution a bit.

Let $dist_i$ denote the shortest distance to point i. Let us also sort the points in increasing height.
We have the following observation:

- Under the above sort, $dist_i \leq dist_j$ for $i \leq j$. This follows immediately from the very first point made, since we can always go to a lower height for no cost.
- Next, let's look at how dijkstra would work on this graph. When at i, we only need to update some vertices greater than i. But, since vertices are sorted by height, we in fact update a contiguous range of indices, starting from $i + 1$ - and all these updates are of the same value (namely, $dist_i + c_i$).

Updates of this form can be done with the help of a sweepline algorithm.
When we are at index i, we need to do the following:

- Find the largest index j such that $h_j \leq h_i + e_i$. This can be done with binary search.
- Now, for each $i + 1 \leq k \leq j$, we need to set $dist_k \leftarrow \min(dist_k, dist_i + c_i)$.
- Looking at it differently, when we are at index i, $dist_i$ is the minimum value over all updates that cover index i.
- So, we can maintain a set of the current updates and the expiry time of the update (i.e, the last position it is valid for, j in the first step). At position i, we do the following:
  - Remove all updates from the set that have already expired before i.
  - Set $dist_i$ to be the minimum value among the updates set.
  - Compute j as described in the first step.
  - Insert $dist_i + c_i$ to the updates set, and set it to expire at $j + 1$

This solves the problem in $O(k \log k)$.

TIME COMPLEXITY

$O(k \log k)$ per test case.

PREREQUISITES:

Single-source shortest path

PROBLEM:

You have an $N \times M$ grid, each cell of which is either blocked or empty. You can travel only between empty cells.

In one move, you can pick any row and unblock all of its cells. Find the minimum number of moves needed so that (N, M) is reachable from $(1, 1)$.

EXPLANATION:

Let's think of the grid as a graph: each cell (i,j) is a vertex, and from (i,j) you can move to the 4 cells $\{(i-1,j), (i+1,j), (i,j+1), (i,j-1)\}$ (if they exist and are empty).

However, this isn't enough information to represent everything we need. In particular, to know whether it's possible to move to another vertex or not, we need two pieces of information:

- Was that cell originally empty?
- If not, has a row bombing operation been performed on its row?

Our graph formulation doesn't encapsulate information about the second point at all, so we need to augment it a bit.

Instead of $N \times M$ vertices, let's consider a graph on $2 \times N \times M$ vertices, where

- $(i,j,0)$ represents cell (i,j) such that a row-bombing operation has *not* been performed on row i.
- $(i,j,1)$ represents cell (i,j) such that a row-bombing operation *has* been performed on row i.

Note that edges now have to be created appropriately when considering the third state, which is a bit of casework.

Details

Consider $(i,j,0)$. From here, we can move to:

- Any of its 4 neighbors mentioned above, in the 0 state, if the corresponding cell was initially empty
- $(i,j,1)$ by bombing this row
- $(i-1,j,1)$ and $(i+1,j,1)$ by bombing the row above/below respectively to clear them out

Similarly, $(i,j,1)$ can move to:

- $(i,j+1,1)$ and $(i,j-1,1)$ freely, since the row has been bombed
- $(i-1,j,0)$ and $(i+1,j,0)$ if the corresponding cells were initially empty
- $(i-1,j,1)$ and $(i+1,j,1)$ by bombing the corresponding rows

This gives us a graph with $\mathrm{O}(N \cdot M)$ vertices and edges.

Note that we somehow want to reach (N, M) from $(1, 1)$ in this graph, while minimizing the number of bombings.
Under our construction, a bombing is performed exactly when we move from a state $(i_1,j_1,0)$ to a state $(i_2,j_2,1)$, i.e, a 0-state to a 1-state.
So, let's weight our graph: every edge from a 0-state to 1-state has weight 1, while every other edge has weight 0.

The answer is then simply the shortest path in this weighted graph from $(1, 1, 0)$ to either $(N, M, 0)$ or $(N, M, 1)$.

This can be computed using Dijkstra's algorithm, of course. However, since the edge weights are only 0 and 1, it's possible to solve the problem in linear time using **0-1 bfs**.

TIME COMPLEXITY

$O(N \cdot M)$ per test case.

PROBLEM:

Given a digital panel and the cost of turning on each of its seven types of segments, find the minimum cost of writing a factor of N on the panel.

EXPLANATION:

N is extremely large, so there is no hope of directly factorizing it and trying to write each factor. We need to be a bit more clever.

First, note that 1 is a factor of any integer. Further, 1 is contained in all of $\{0, 3, 4, 7, 8, 9\}$ (in terms of segments that need to be drawn).
So, instead of writing any of these on the panel, we could just write 1 instead for strictly lower cost. They can thus be discarded entirely.

That leaves us with $1, 2, 5, 6$.

A bit more analysis tells us:

- Using 1 in any number with length > 1 is pointless, we could just write the single digit 1 instead.
- If the number we write ends with 2 or 5, then we could just write the single digit 2 or 5 respectively for lower cost while still maintaining divisibility.
- So, if we are to write a number with length > 1, it must end with 6 and won't contain 1.
- A number ending with 6 is even, so if such a number contains a 2 somewhere, it's better to just write the single digit 2.

So, for numbers with length > 1, we only need to care about those that end with 6 and use the digits 5 and 6. How many of these do we need to check?

As it turns out, not too many!

Note that the costs are rather small, so it quickly becomes optimal to just write 1 instead of some long number consisting of 5's and 6's.

In particular, if we set $B = 50$ and all the other costs to 1, then writing 1 costs 51, writing 5 costs 5, and writing 6 costs 6. This is essentially the worst case, since increasing the cost of 1 would increase the cost of 5 and 6 by the same amount (all three share the C segment).

So, writing any length 10 or longer number is not optimal: it is better to just write 1.

There are $\leq 2^{10}$ numbers of length < 10 whose digits consist of 5 and 6, so test them all. Checking whether a small number x divides N can be done in $O(|N|)$ by iterating across the digits of N and maintaining the remainder modulo x, and doesn't need a BigInteger class. You can also just use Python.

Don't forget to also check with 1 and 2. Finally, print the minimum across all factors.

PREREQUISITES:

Frequency arrays, Basic combinatorics

PROBLEM:

You are given an array A and an integer K. Count the number of subsequences that don't contain any pair whose sum is divisible by K.

EXPLANATION:

First, notice that the condition "$A_i + A_j$ is divisible by K" can be written as $A_i + A_j \equiv 0 \pmod K$.

In particular, we can work with all the array elements modulo K so that they're all between 0 and $K-1$.

Now, consider what happens when we have $x + y \equiv 0 \pmod K$ when both x and y are less than K. There are three possibilities:

- First, we can have $x = y = 0$
- Second, if K is even we can have $x = y = K/2$
- Finally, if neither of the above hold, we must have $y = K - x$; and in particular $x \neq y$.

Let's leave the first two cases alone for now, and look at the third.
For convenience, let $x < K - x$.
Note that for each x, any good subsequence can have *either* some occurrences of x, *or* some occurrences of $K - x$: never both.
In particular, we can take as many x-s as we like, or as many $(K - x)$-s as we like, without affecting any other sums (since $K - (K - x) = x$). Essentially, we 'pair up' x with $K - x$, and then different pairs are completely independent.

So, the choices of which of the x's or $(K - x)$'s we take are completely independent across different x.
This means that any subsequence can be constructed as follows:

- Choose a subset of 1's *or* a subset of $K - 1$'s
- Then, choose a subset of 2's *or* a subset of $(K - 2)$'s
- Then, choose a subset of 3's *or* a subset of $(K - 3)$'s

⋮

Thus, the total number of subsequences can be found by multiplying the number of choices for different x.

This brings us to the questions: how many choices are there for a fixed x?

Answer

Let $freq(x)$ be the number of occurrences of x in the array.

Note that we can choose any subset of the x's, or any subset of the $(K - x)$'s.
The first one gives us $2^{freq(x)}$ choices, while the second gives us $2^{freq(K-x)}$ choices.

The empty set is counted in both, so we need to subtract 1 to avoid overcounting.
This brings the total to $2^{freq(x)} + 2^{freq(K-x)} - 1$.

The number of subsequences is thus just the product of $(2^{freq(x)} + 2^{freq(K-x)} - 1)$ across all x such that $x < K - x$.

The only exceptions here are $x = 0$ and (if K is even) $x = K/2$, which shouldn't be included in the above product because they behave slightly differently. Do you see how to deal with them?

Answer

$x = 0$ and $x = K/2$ follow a simple rule: there can't be more than one of each in the subsequence.

So, we have $1 + freq(0)$ choices for 0 (choose none of them, or choose exactly one), and similarly $1 + freq(K/2)$ choices for $K/2$.

Multiply these quantities to the previous value to obtain the final answer.

Notice that the value for a given x requires us to compute a power of 2 modulo something.
There are several ways to do this: the simplest is to just precompute the value of $2^x \pmod{MOD}$ for every $0 \le x \le 5 \cdot 10^5$ before processing any test cases, after which these can be used in $O(1)$.
Alternately, you can use **binary exponentiation**.

TIME COMPLEXITY

$O(N + K)$ per test case.

PROBLEM:

Given an array A of length N.

You are allowed to perform the following operation on the array A **atmost** 20 times:

- Select a **non-empty** subset S of the array $[1, 2, 3, \ldots, N]$ and an integer X $(0 \leq X \leq 10^6)$;
- Change A_i to $A_i + X$ for all $i \in S$.

You have to sort the array A in **strictly increasing** order by performing **atmost** 20 operations.

It is guaranteed that we can always sort the array A under given constraints.

EXPLANATION:

Let array $B = [10^5, 10^5 + 1, ..., 10^5 + N - 1]$. Array B is **strictly increasing** . No matter what the initial array A is we can always change it to B in ≤ 20 steps. Construct a new array $Diff$ of length N such that $Diff_i = B[i] - A[i]$. It is obvious we need to add $Diff_i$ to each A_i to make it equal to B_i. The maximum value of $Diff_i$ does not exceed $2 * 10^5$. This means only bits till 0 to 17 can be set to 1 in the binary representation of any number $Diff_i$. In each operation from $i = 0$ to 17 choose all indices j in $Diff$ such that i^{th} bit is set to 1 in $Diff_j$ and add 2^i to all such indices in A. By the end of 18^{th} operation array A is transformed to array B.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Knowledge of the binary number system

PROBLEM:

Kulyash has an integer N. In one operation, he can take an integer X he has and break it into two integers Y and Z whose sum is X. Find the minimum number of operations needed so that every integer with him is a power of 2.

EXPLANATION:

Note that each operation performed doesn't change the sum of all the numbers Kulyash has. In particular, the sum is always going to be N.

At the final state, when everything is a power of 2, this means that we have written N as the sum of some powers of 2. So, if we are able to find the minimum number of powers of 2 needed to write N, we will then be able to find our answer — if we need K powers of 2 to sum up to N, the answer is $K-1$ because we can create one of these powers at each step (and the last step will create two).

So, what is the minimum number of powers of 2 needed?

Answer

Write N in binary. If it has K set bits, then we need at least K powers of 2 to sum up to N.

The proof of this should be fairly easy to see if you are familiar with the binary number system, but is included below for completeness.

Proof

Suppose we write $N = x_1 + x_2 + \ldots + x_m$, where each x_i is a power of 2. If some two of the x_i are equal, we can combine them into a single larger power of 2 and obtain a sum of N with $m-1$ powers of 2 instead.

Hence, when N is written as the sum of the lowest possible number of powers of 2, all these powers must be distinct. However, by the property of the binary number system, there is **exactly one** way to write a number as the sum of distinct powers of 2.

This completes the proof.

Once we know the above, the implementation is fairly simple. Count the number of set bits in the binary representation of N (either by looping over each bit or using some inbuilt function), let this be K. The answer is then $K-1$.

TIME COMPLEXITY:

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

PREREQUISITES:

Two pointers or deques

PROBLEM:

Given an array A, in one move you can pick any A_i and split it into X and Y such that $X + Y = A_i$.
Find the minimum number of moves to make A a palindrome.

EXPLANATION:

Let's look at the two end elements, i.e, A_1 and A_N.
If $A_1 = A_N$, we can ignore them and continue on with the rest of the array: we now deal with $N - 2$ elements.

Otherwise, suppose $A_1 < A_N$. Note that we have to perform at least one move on A_N to make the endpoints equal.
So, let's perform this move: split A_N into $(A_N - A_1)$ and A_1.
Now the endpoints of the array are equal, so we can continue on with the rest of the array. However, notice that we added the new element $A_N - A_1$ to the array, which we also need to take care of. So, we deal with $N - 1$ elements now.
The $A_1 > A_N$ case can be handled similarly: we just need to insert $A_1 - A_N$ at the start of the array instead.

In either case, the array length decreases by at least 1, so the process will be done at most N times before we end. If we can simulate this algorithm fast enough, we are done.

Implementing this by deleting from arrays/vectors/etc will lead to a solution in $\mathrm{O}(N^2)$ because it's simply not possible to easily insert/delete elements at the beginning.

However, there are a couple of ways to overcome this:

- Perhaps the simplest way is to just use a data structure that *does* allow for quick insertion/deletion at both ends: a deque.
  - With a deque, implementation becomes extremely easy: while there is more than one element, compare the front and back elements, then insert the appropriate new element to either the front or back.
- It's also possible to implement this using two pointers (which really just simulates a deque on an array anyway):
  - Start with $L = 1, R = N$.
  - At each step, compare A_L and A_R.
  - Deleting elements can be simulated by increasing L/decreasing R
  - Adding a new element at the front/back can be done by decreasing L/increasing R and then setting A_L or A_R appropriately.

TIME COMPLEXITY

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

PROBLEM:

For a binary string A, let $f(A)$ denote its *badness*, defined to be the difference between the number of zeros and number of ones present in it. That is, if the string has c_0 zeros and c_1 ones, its badness is $|c_0 - c_1|$. For example, the badness of "01" is $|1 - 1| = 0$, the badness of "100" is $|2 - 1| = 1$, the badness of "1101" is $|1 - 3| = 2$, and the badness of the empty string is $|0 - 0| = 0$.

You are given an integer N and a binary string S.

You would like to partition S into K disjoint **subsequences** (some of which may be empty), such that the maximum badness among all these K subsequences is **minimized**. Find this value.

Formally,

- Let $S_1, S_2, \ldots, S_K$ be a partition of S into disjoint subsequences. Every character of S must appear in one of the S_i. Some of the S_i may be empty.
- Then, find the *minimum* value of $\max(f(S_1), f(S_2), \ldots, f(S_K))$ across all possible choices of $S_1, S_2, \ldots, S_K$ satisfying the first condition.

EXPLANATION:

Let there be O ones and Z zeros in S. After dividing S into K boxes let there be Z_i zeros and O_i ones in box i.

We can write, $Z - O = (Z_1 - O_1) + (Z_1 - O_1) + ...(Z_K - O_K)$.

If $Z > O$ then it is always better to divide S so that for each of the K boxes $Z_i \geq O_i$. This is because if for some box x we have $Z_x < O_x$, there has to be at least one box y having $Z_y > O_y$ because $Z > O$ overall. Reducing 1 zero from box x and placing it in box y instead would have helped to reduce the badness for both boxes x and y by 1. This is clearly not optimal. Using the above observation it is clear that $Z_i - O_i$ is the badness for box i. Hence, we need to minimize the maximum among $(Z_i - O_i)$ and this is achieved when all the K terms have as close values as possible and add up to $Z - O$ which is known. The maximum term after such division is $\lceil \frac{Z-O}{K} \rceil$, which is our answer.

For $Z < O$ the vice versa (replace ones by zeros and zeros by ones) is true and the answer is $\lceil \frac{O-Z}{K} \rceil$ and if $Z = O$ then the answer is 0 because we can place the entire S into 1 box.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

BFS, Dynamic programming

PROBLEM:

You have N integers. In one move, you can choose A_i and set it to either A_i^2 or $\lfloor\sqrt{A_i}\rfloor$.

Find the minimum number of moves required to sort A.

EXPLANATION:

The single most important observation here is that a given index can't take too many values: for the given limits, there are ≤ 40 possibilities for each index.

Proof

We can make a couple of observations for any positive integer x.

- $\lfloor\sqrt{x^2}\rfloor = x$. This means that any sequence of operations on x can be replaced by an equivalent sequence of operations in which *every square root operation is before every squaring operation*.
- If we don't use the square root operation, the integers we can reach are $x, x^2, x^4, x^8, \ldots$, i.e, integers of the form x^{2^k} for some $k \geq 0$.

If $x = 1$ the only integer that can be reached is 1, so we only look at $x > 1$.
If $x > 1$, then note that $x^{2^6} = x^{64} > 10^{18}$ no matter what x is. So, repeated squaring only allows us to reach at most 6 values: $x, x^2, x^4, x^8, x^{16}, x^{32}$.

Similar reasoning should tell you that repeatedly taking the square root of x will give you at most 6 distinct values before you hit 1.
Since any valid integer can be created by taking the square root some times and then squaring some times, this gives us an upper bound of $6 \times 6 = 36$ reachable values.

Further, while it isn't needed, a slight modification of the argument allows us to prove a tighter bound: $6 + 5 + 4 + 3 + 2 + 1 = 21$, which along with $x = 1$ gives us 22. Do you see how?

With this in mind, let's first find, for each $1 \leq i \leq N$, the possible values that can occur at this position (and the minimum number of steps needed to reach each value).
Let S_i denote the set of values that can occur at position i, and $cost(i, x)$ denote the number of steps needed to turn A_i into x.

How to compute these?

We use bfs!

More specifically, consider the (infinite) directed graph on the set of positive integers, where there's an edge from x to y if and only if $y = x^2$ or $y = \lfloor\sqrt{x}\rfloor$.

Then, note that S_i is exactly the set of nodes that can be reached from A_i in this graph, and $cost(i, x)$ is simply the shortest path from A_i to x.
So, simply start a bfs from A_i to compute distances: while the graph itself is impossible to hold in memory, we only care about those vertices that can be reached from A_i, which as noted above is pretty small. Edges are defined implicitly and can be computed as we do the bfs.

This bfs will take $O(|S_i|)$ time for a given index i, and $|S_i| \leq 40$ so this is $\leq 40N$ vertices visited in total, which is perfectly fine.

Once we have this information, the rest of the problem turns into a relatively simply dp.

Let $dp_{i,x}$ denote the minimum number of moves needed to sort the first i elements of the array, such that the value at position i is x. The final answer is the minimum value of $dp_{N,x}$ across all $x \in S_N$.

The transitions are simple:

$$dp_{i,x} = cost(i,x) + \min_{\substack{y \in S_{i-1} \\ y \le x}} (dp_{i-1,y})$$

Or in simpler words: the minimum cost to sort the first i elements and have x end up at position i equals the cost of turning A_i into x, plus the minimum cost of sorting the first $i-1$ elements such that position $i-1$ contains an element not larger than x.
The first part is $cost(i,x)$ which we precomputed, and the second part is exactly what our dp contains so we obtain a recursion.

Let M be the maximum number of possibilities for a single index. Note that $M \le 22$.
The algorithm above, if implemented directly as described, uses $O(NM)$ space and has a worst-case complexity of $O(NM^2)$.

It is possible to improve both of these:

- Note that computing $dp_{i,x}$ depends on only the values of dp_{i-1}. So, we can discard all previous rows of the dp table, which brings the memory down to $O(N+M)$.
- Computing the transitions itself can be improved to $O(NM)$ by sorting the sets S_i and iterating in increasing order to compute dp_i, and then maintaining two pointers (or with binary search for an additional $\log$ factor, which should still be fast enough).

The second step is explained more in-depth in the editorial of **DIVSORT** (which is a pretty similar problem, and has essentially the same dp).

If your solution matches the above and you're getting WA, it might be due to use of the `sqrt` function: read through **this blog** to see how to fix it.

TIME COMPLEXITY

$O(NM)$ or $O(NM \log M)$ per test case, where $M \le 22$ is the maximum possible number of values an index can have.

PREREQUISITES

Strings, Greedy

PROBLEM

You are given a binary string S of length N. You can reverse any substring of S in one operation, find the minimum number of operations required to sort S.

EXPLANATION

Hint

The resulting string should be such that all '0's appear consecutively before a '1' appears.

Detailed Explanation

To sort S, we can iterate through the string and reverse every leftmost instance of "$1, 1, ..., 1, 0, ..., 0, 0$", i.e. a substring of S with consecutive '1's and then consecutive '0's. This is guaranteed to sort S in the minimum number of operations.

Intuition for correctness

While iterating through S, if you reverse an instance of "$1, 1, ..., 1, 0, ..., 0, 0$" then you would obtain a prefix of '0's, as we are essentially *pushing forward* the occurrences of '1's that are present before a '0'. So, the resulting string would have a prefix of '0's and a suffix of '1's, and hence would be sorted in the minimum number of operations.

This simply reduces to counting the number of occurrences of "10" that appear in the string and printing it. This is because the number of "$1, 1, ..., 1, 0, ..., 0, 0$"s that appear in the string are the same as the number of "10"s that appear in the string.

TIME COMPLEXITY

The time complexity is $O(|S|)$ per test case.

PROBLEM:

You have a binary string S of length N. In one move, you can delete any sorted substring of S. Find the minimum number of deletions to obtain a sorted string.

EXPLANATION:

Deleting a sorted substring means we delete either:

- A substring whose characters are all the same; or
- A substring that looks like $000\ldots000111\ldots111$

This also tells us that the final string must be of this form. In particular, a binary string is sorted if and only if it doesn't contain 10 as a subsequence, so our aim is to remove all such subsequences.

We can now make a few observations:

- One simple observation we can make is that if $S_i = S_{i+1}$ for some index i, then in any solution either both these characters must be deleted, or they both can be left alive. (do you see why?)
  - This allows us to essentially 'compress' the string by considering only adjacent unequal characters, i.e, make the string look like either $101010\ldots$ or $010101\ldots$
- From now on, we'll only consider such compressed strings. Note that they will be alternating.
- If the first character is 0, we can just leave it alone and never delete it, since it'll never affect sortedness. So, we can assume that the first character is always 1.
- Similarly, if the last character is 1, we can leave it alone. So, we can assume the last character is always 0.

Now note that we are in a case where we have an alternating binary string, whose first character is 1 and last character is 0. This means it must look like $1010\ldots10$, and in particular has even length, say $2K$.
The minimum number of moves needed to sort such a string is simply K.

Proof

K is clearly an upper bound — you can use K moves to remove all the K ones, leaving only the zeros which are sorted.

On the other hand, we also always need at least K moves.
Let's break the string into K substrings, each of the form 10. Note that either a 1 or a 0 (or potentially both) must be deleted from each of these substrings: leaving both alive would make the final string non sorted.

Now, let's look at our possible moves. Since the substring chosen must be sorted, we have only 3 options:

- Choose a 0
- Choose a 1
- Choose a 01

If we choose a 01, we simply end up in the same situation but with length $2K-2$ instead.
So, let's look at the case when we can only delete a 0 or a 1.

As noted above, we have K substrings of the form 10 that all need to be broken. With one character being deleted per move, we obviously need at minimum K moves to break all these substrings, giving us our lower bound.

The final solution is thus to compress the input string into an alternating binary string, delete the first character (if it is 0) and/or last character (if it is 1), and then print half the length of the obtained string.

Note that this is also simply the number of times 10 appears as a substring of S, which is an easy way to implement the solution.

PREREQUISITES:

Sorting, (optional) Range-min data structures

PROBLEM:

There are N items, the i-th of which has weight W_i and value G_i. Upon arranging these items into a binary search tree, the key of the i-th element K_i equals its value, plus the sum of weights of all its descendants (excluding itself).

Across all possible arrangements into a BST, find the minimum possible value of $\max_{i=1}^{N} K_i$.

EXPLANATION:

Let's determine what the optimal choice of root is. Since the weights are distinct, choosing the root will immediately partition the vertices into two sets based on their weights, and we can try to solve the problem on these reduced sets.

Computing its key, we see that

$$K_{root} = G_{root} + \sum_{\substack{i=1 \\ i \neq root}}^{N} W_i = (G_{root} - W_{root}) + \sum_{i=1}^{N} W_i$$

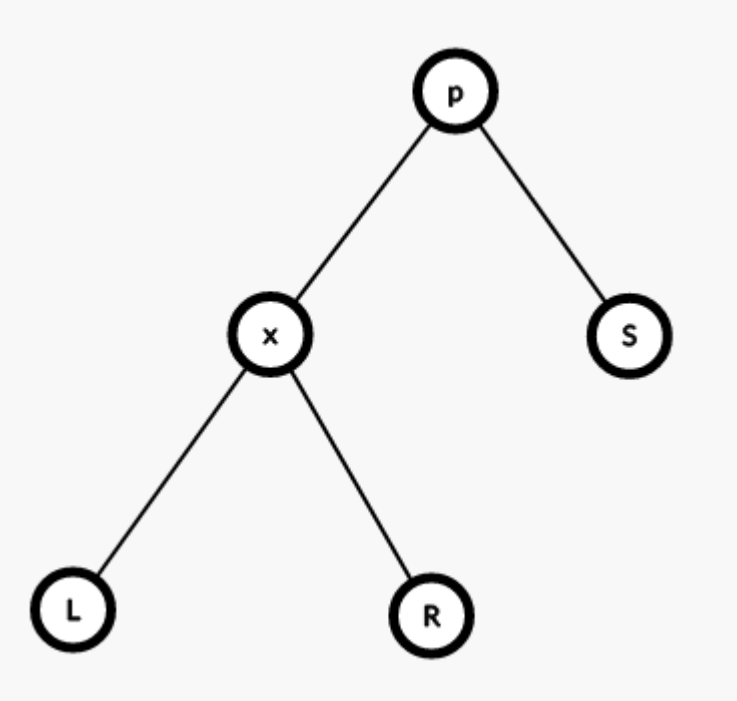
Note that $\sum_{i=1}^{N} W_i$ is a constant. In other words, the key value of the root is minimized by simply choosing whichever element u has the lowest value of $G_u - W_u$.

It turns out this choice is optimal! This means that we can simply find such a u, make it the root, then find the sets of vertices on its left and right and solve for them recursively.

Proof

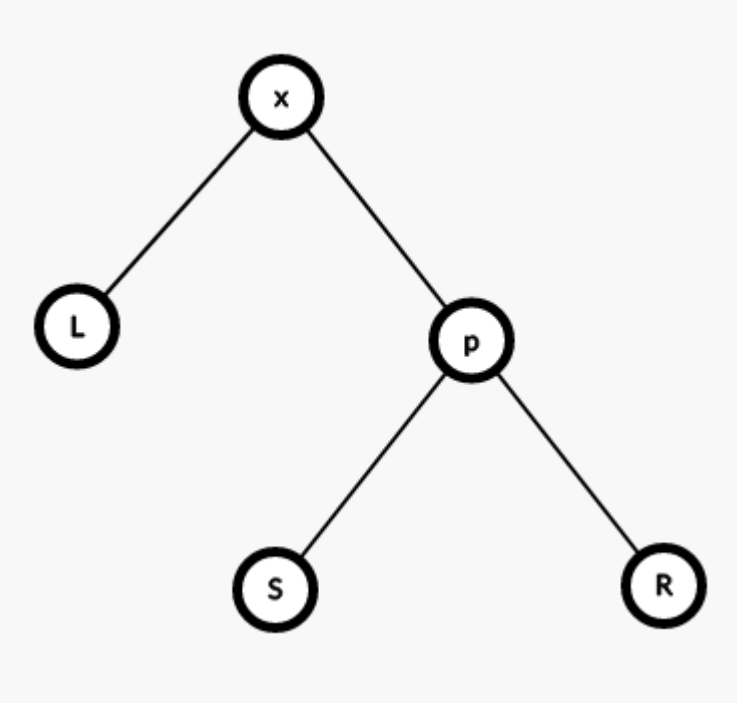
Consider some element x, and let p be its parent. Our claim simply boils down to the fact that $G_p - W_p \leq G_x - W_x$.

So, suppose this weren't the case, and $G_p - W_p > G_x - W_x$.
Without loss of generality, let x be the *left* child of p (the proof for when it is the right child is similar).
Essentially, we have something like the image below:



Here, S, L, R all represent subtrees and not single nodes.

Let's rotate the BST so that x is now the parent of p. In order to keep the BST structure, we will end up with the following:



Now, note that:

- The key values of any vertices not shown in the diagram don't change at all.
- The key values of any vertices lying in the subtrees S, L, R don't change at all
- The new key value of x is strictly less than the old key value of p, since $G_x - W_x < G_p - W_p$.
- The new key value of p is strictly less than the old key value of p since it's now the root of a smaller subtree.

So, the maximum K_i in the new tree is no larger than the maximum K_i in the old tree, which is exactly what we wanted.

It's easy to see that only a finite number of rotations of this form can be made, so eventually we will end up with a tree of the form described above, proving optimality of our algorithm.

However, directly implementing this gives us a solution in $O(N^2)$ — we use $O(L)$ time to solve for a set of size L and then recurse to its children, so if the tree degenerates to a line this comes out to be $O(N) + O(N-1) + O(N-2) + \ldots = O(N^2)$.

There are a couple of ways to optimize the solution, but they all depend on a simple observation: suppose we sort the elements in ascending order of W_i. Then, any set we solve for is a subarray of this sorted array.

Proof

This should be pretty easy to see, and can be proved inductively.
We start off with the entire range $[1, N]$. Once we choose a root r, we recurse into the weights that are $< W_r$ and $> W_r$. These are exactly the ranges $[1, r-1]$ and $[r+1, N]$.
Now we apply the same process to these smaller ranges, and so on.

So, let's sort the elements by W_i.

Now, to solve for a range $[L, R]$, we need to:

- Find $L \le u \le R$ such that $G_u - W_u$ is minimum in this range
- Compute the key value of u
- Recurse into $[L, u-1]$ and $[u+1, R]$.

The first and second steps are what consume the most time. However, note that they simply require range-min and range-sum respectively.

So, build some range-min data structure on the $G_i - W_i$ values, and also the prefix sum array of W, say P. Then,

- The first step can be done in $O(1)$ or $O(\log N)$ depending on the structure used (sparse table/segment tree), by simply finding $rangemin(L, R)$.
- The second step can be done in $O(1)$ by computing $rangesum(L, R)$

The runtime is thus $O(N)$ or $O(N \log N)$, since we perform an $O(1)/O(\log N)$ operation at every vertex of an N-sized tree.

It is also possible to solve the problem without any data structures at all. Note that the tree we build after sorting is nothing but the **Cartesian tree** of the $G_i - W_i$, and building such a tree can be done in linear time — methods to do so are described in the link.

TIME COMPLEXITY

$O(N \log N)$ per test case.

PREREQUISITES:

Heaps

PROBLEM:

You are given an integer N. You have to find a permutation of length N that satisfies M conditions of the following form:

- (X_i, Y_i), denoting that the element X_i $(1 \leq X_i \leq N)$ must appear in the prefix of length Y_i. Formally, if the index of the element X_i is K_i, then the condition $1 \leq K_i \leq Y_i$ must hold.

print `-1` if no such permutation exists. In the case of multiple permutations, find the **lexicographically smallest** one.

If two arrays A and B have the same length N, then A is lexicographically smaller than B only if there exists an index i $(1 \leq i \leq N)$ such that $A_1 = B_1, A_2 = B_2, \ldots, A_{i-1} = B_{i-1}, A_i < B_i$.

EXPLANATION:

Hint

Try to construct the permutation from back to the front.

Solution

Using the M given conditions we can directly construct a list for each index containing all the elements which must appear at that index or before it. All the elements which don't have any condition associated with them will appear in the list corresponding to index N. Let us call these collection of lists as V.

Since we want to construct the lexicographically smallest permutation, we can always fill the permutation from the back to the front, starting from the largest elements possible first. It is clear that at the last index (N) only the elements in V in the list corresponding to index N can appear. We will fill the last index with the largest element allowed. This greedy logic can be continued till the first index. We will maintain a priority queue which stores all the allowed elements at any point and this makes it possible to extract the largest element easily. We will keep on adding elements to this priority queue using V as we go from index N to 1.

At any point, if the priority queue is empty this means that there are no elements allowed at that index. This means that the permutation cannot constructed and the answer is -1.

TIME COMPLEXITY:

$O(NlogN)$ for each test case.

PREREQUISITES:

Observation, the **Sprague-Grundy Theorem and Nim**

PROBLEM:

You have an array A of **distinct** integers. Define its *goodness* to be $\sum_{i=2}^{N} |A_i - A_{i-1}|$.

Alice and Bob alternate moves, each player on their turn deleting a non-empty subarray from A that doesn't change its goodness. Under optimal play, who wins?

EXPLANATION:

There is a single important observation that allows for a solution to this problem:

Suppose a player deletes a subarray $[L, R]$. This doesn't change the goodness of A if and only if:

- $A_{L-1} < A_L < A_{L+1} < \ldots < A_R < A_{R+1}$, or
- $A_{L-1} > A_L > A_{L+1} > \ldots > A_R > A_{R+1}$

That is, the subarray $[L-1, R+1]$ must be either increasing or decreasing. In particular, $L \geq 2$ and $R \leq N-1$ must also hold.

Proof

It is easy to see that deleting a subarray of the above form doesn't change the goodness of the array — both before and after deleting, the contribution of the $[L-1, R+1]$ section is $|A_{L-1} - A_{R+1}|$, and the contribution of indices before $L-1$ and after $R+1$ isn't affected in any way.

It is also easy to see that deleting a subarray with $L = 1$ or $R = N$ always reduces the goodness: some existing pairs stop contributing, and no new pairs contribute.

Now, consider a subarray $[L, R]$ such that $[L-1, R+1]$ is not sorted. Without loss of generality, let $A_{L-1} < A_{R+1}$.
Since the subarray is not sorted, there exists an index i such that $L \leq i \leq R+1$ and $A_i < A_{i-1}$. Choose the first such index i.

Now,

- After deletion, this segment contributes $A_{R+1} - A_{L-1}$ to the goodness, since those elements are now adjacent.
- Before deletion, the goodness was at least $(A_{i-1} - A_{L-1}) + (A_{i-1} - A_i) + |A_{R+1} - A_i|$
  - If $A_{R+1} > A_i$, this is $(A_{R+1} - A_{L-1}) + 2A_{i-1} - 2A_i$, which is strictly larger than $A_{R+1} - A_{L-1}$ since $A_{i-1} > A_i$.
  - If $A_{R+1} < A_i$, this is $2A_{i-1} - A_{L-1} - A_{R+1}$, which equals $(A_{R+1} - A_{L-1}) + 2A_{i-1} - 2A_{R+1}$, which is once again strictly larger than $A_{R+1} - A_{L-1}$.

So, deleting such a subarray can never keep the goodness equal.

Now, note that the initial array A can be split into alternating increasing and decreasing subarrays of length ≥ 2 that intersect only at their borders, and performing a deletion within each of these subarrays is independent because the borders cannot be deleted.

For example, $A = [1, 5, 3, 2, 4]$ splits into $[1, 5], [5, 3, 2], [2, 4]$.

Consider one such segment, of length x. In one move, a player can delete some contiguous elements from it: in particular, anywhere between 1 and $x-2$ elements can be deleted.

This is strikingly similar to the classical game of nim: indeed, this subarray corresponds to a nim pile of size $x - 2$.
Our piles are independent, and so we can simply apply the solution to nim here:

- Find all the alternating increasing/decreasing subarrays as mentioned above. Let their lengths be $x_1, x_2, \ldots, x_k$.
- Then, we have nim piles of sizes $x_1 - 2, x_2 - 2, \ldots, x_k - 2$.
- So, Bob wins if and only if $(x_1 - 2) \oplus (x_2 - 2) \oplus \ldots \oplus (x_k - 2) = 0$, where $\oplus$ denotes the bitwise xor operation.

Finding the maximal increasing/decreasing subarrays can be done in $\mathrm{O}(N)$ in several ways, for example with two-pointers.

TIME COMPLEXITY

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

PROBLEM:

You have a boolean array A. In one move, you can choose any subarray of A with length at least 2, add its bitwise XOR to your score, and delete this subarray from A.

Find the maximum possible score you can achieve.

EXPLANATION:

Since the array is boolean, each move increases our score by either zero or one.
Note that making a move that increases our score by zero is pointless, so we really want to count the maximum number of 1 moves that we can make.
Let's call a subarray with xor 1 a *good* subarray.

Further, it's better to use shorter subarrays if possible, since that gives us more freedom in the future. The shortest possible good subarrays we can choose are $[0, 1]$ and $[1, 0]$, so let's keep choosing these as long as its possible to do so.

Suppose we can't choose any more subarrays of this kind. Then, every element of A must be the same, i.e, A consists of all 0's or all 1's.

- In the first case, all 0's, nothing more can be done, since any remaining subarray has xor 0.
- In the second case, we still have good subarrays: anything with odd length is good, i.e, $[1, 1, 1], [1, 1, 1, 1, 1], \ldots$
  - Suppose the length of A is now K. Then, the best we can do is $\lfloor \frac{K}{3} \rfloor$ subarrays, each of the form $[1, 1, 1]$. So, add this value to the answer.

This gives us the final solution:

- While the array contains both 0's and 1's, remove one 0 and one 1 from it, and increase the answer by 1
- When the array contains only a single type of character, if it's 1 and there are K of them, add $\lfloor \frac{K}{3} \rfloor$ to the answer.

This can be done in $\mathrm{O}(1)$ by knowing the counts of 0's and 1's, although simulation in $\mathrm{O}(N)$ will still pass.

TIME COMPLEXITY

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

PROBLEM:

Given a binary string S, in one move you can choose any substring of S that has an equal number of zeros and ones, and replace it with a single 0 or 1.
Find the minimum length of the final string, and a sequence of moves that achieves this.

EXPLANATION

First, let's get one edge case out of the way: if S consists of only a single character, no moves can be made on it so the minimum length is just N.

In any other case, you can bring the length down to 1.
A proof of this also gives us a construction.

Let d denote the difference between the number of ones and the number of zeros in the string.

- If $d = 0$, the string has an equal number of zeros and ones, so just replace the entire string with a single character.
- Otherwise, suppose $d > 0$ (the $d < 0$ case can be handled similarly). This means there are d more ones then zeros.
  - S has at least one 1 and one 0 (since the case when it doesn't was taken care of right at the start).
  - In particular, it has at least one substring that is either 10 or 01.
  - Replace this substring with a 0 (if $d < 0$, replace it with 1 instead).

Notice that doing this operation reduces d by 1 (since we effectively just deleted a 1 from the string and did nothing else), while also ensuring that the string still has both 1's and 0's. So, this can be repeated till we reach $d = 0$, at which point the entire string is replaced.

The small limit on N allows for each operation to be done in $O(N)$, giving us a $O(N^2)$ solution. However, it also possible (and not very hard) to implement this in $O(N)$.

TIME COMPLEXITY

$O(N^2)$ or $O(N)$ per test case, depending on implementation.

PROBLEM:

Chef has an array A of length N.

In one operation, Chef can choose any **subsequence** of the array and any integer X and then add X to **all** the elements of the chosen subsequence.

Determine the **minimum** number of operations required to make all the elements of the array **distinct**.

EXPLANATION:

Suppose in the array there are distinct elements $A_1, A_2 \ldots A_n$, with frequencies $f_1, f_2, ..., f_n$. Then for each element A_i, we will proceed as follows:
$Case\ 1 : f_i = 1$
In this case the number is already distinct, so no operation is required.
$Case\ 2 : f_i > 1$
In this case we will select half of elements of A_i and increase it by an arbitrary high number.

We can do the operations for all the distinct elements at the same time.
So in each operation, we will select all the distinct elements A_i, such that $f_i > 1$ and increase half of the elements to a arbitrary higher value.
Thus in all we just have to calculate the maximum f_i, say f_{max} and in each operation we will reduce the frequency f_{max} by half, thus our answer would be

$$\lceil log_2 f_{max} \rceil$$

Since operations would be applied to all $f_j < f_{max}$, at the same time so we just have to calculate for maximum f_i.

TIME COMPLEXITY:

$O(N)$, for each test case.

PROBLEM:

A *segment* is a range of **non-negative** integers $L, L+1, L+2, \ldots, R$, denoted $[L, R]$ where $L \leq R$.

Chef has a set S consisting of **all** segments $[L, R]$ such that either $L + R = X$ or $L \cdot R = Y$.

For example, if $X = 5$ and $Y = 6$, then Chef's set is $S = \{[0, 5], [1, 4], [1, 6], [2, 3]\}$.

Given the integers X and Y, can you help Chef find two non-intersecting segments from his set S? If it is not possible to find two such non-intersecting segments, print -1. If there are multiple possible answers, you may output **any** of them.

Note: Two segments are said to be non-intersecting if they have no number in common. For example, $[1, 4]$ and $[10, 42]$ are non-intersecting, while $[1, 4]$ and $[4, 6]$ are not since they have 4 in common.

EXPLANATION:

Observation 1: The smallest interval $[L, R]$ such that $L + R = X$ would be:

$$[\frac{X}{2}, \frac{X}{2}], \ \textit{if } X \textit{ is even.}$$

$$[\frac{X}{2}, \frac{X}{2} + 1], \ \textit{if } X \textit{ is odd.}$$

Any other interval would overlap the above one.

So we can choose one of the intervals as above and then loop through all possible intervals such that $L \times R = Y$ and for each interval we check if the two intersect each other or not.

TIME COMPLEXITY:

$O(\sqrt{N})$, for each test case.

PREREQUISITES:

(Optional) Deques

PROBLEM:

Alice and Bob play a game on a binary string S, creating a new string T. They alternate moves, with Alice going first.

- Alice takes the first remaining character of S and moves it to either the start or end of T
- Bob takes the last remaining character of S and moves it to either the start or end of T

Alice tries to (lexicographically) minimize T, while Bob tries to maximize it. What is the final string?

EXPLANATION:

Alice and Bob end up having extremely well-defined moves.

- Since Alice wants to minimize the string, she will always move a 0 to the front and a 1 to the end of T.
- Conversely, Bob will always move a 0 to the back and 1 to the front of T.

This is already enough to solve the problem in $\mathrm{O}(N^2)$ by directly simulating each move:

- Let T be an empty string.
- On Alice's turn:
  - If the character is 0, insert it to the front of T
  - If the character is 1, insert it to the back of T
- On Bob's turn:
  - If the character is 1, insert it to the front of T
  - If the character is 0, insert it to the back of T

Finally, print T.

For the limits given in the problem, this is already good enough. However, with the help of data structures, there exists a faster solution.

Knowing the moves, all that needs to be done is to simulate the game quickly. For this, we would like a data structure that allows us to quickly insert elements at both the start and the end. The appropriate structure here is a `deque` (`std::deque` in C++, `collections.deque` in Python).

The final solution is thus:

- Keep a deque T, which is initially empty.
- On Alice's turn:
  - If the character is 0, push it to the front of T
  - If the character is 1, push it to the back of T
- On Bob's turn:
  - If the character is 1, push it to the front of T
  - If the character is 0, push it to the back of T

Finally, print T from front to back.

Every operation on T is $\mathrm{O}(1)$, and by choosing the first/last character of S by maintaining two pointers (instead of directly deleting the character) these operations also become $\mathrm{O}(1)$. This leads to a final complexity of $\mathrm{O}(N)$.

PROBLEM:

You are given 2 integers N and M. You also have an empty set S.

Consider an array A of length N such that $1 \leq A_i \leq M$ for each $1 \leq i \leq N$ (there are M^N such arrays in total).

For each of these M^N arrays, insert **all** of its longest non-decreasing subsequences into S.

Output the size of S. Since the size of S might be large, output it modulo $10^9 + 7$.

Note that since S is a set, it stores only distinct values. For example, when $N = 3$ and $M = 2$, the sequence $[1, 1]$ is a longest non-decreasing subsequence for both $A = [1, 2, 1]$ and for $A = [2, 1, 1]$ — however, it will only be present once in S.

EXPLANATION:

Observation: All the non-decreasing sequences greater than or equal to the length of $\lceil \frac{n}{m} \rceil$ belongs to the set.

Here since we only have m numbers so some number would atleast appear $\lceil \frac{n}{m} \rceil$ times so minimum length of non-decreasing sequences would be $\lceil \frac{n}{m} \rceil$.

Hence our answer would be the count of all non-decreasing sequences of length $\geq \lceil \frac{n}{m} \rceil$ since some element would appear with frequency $\geq \lceil \frac{n}{m} \rceil$, thus given any LIS of length greater than or equal to this, it's possible to construct a full sequence.

Thus to count the number of non decreasing sequences, we just need to fix the frequencies of each element. In order to do this we can represent this as a multiplication of polynomials.

$$\underbrace{(1 + x^1 + x^2 ....)}_{x^i \text{ denotes using } i \text{ units of 1}} \times \underbrace{(1 + x^1 + x^2 ....)}_{x^i \text{ denotes using } i \text{ units of 2}} \times ..... \times \underbrace{(1 + x^1 + x^2 ....)}_{x^i \text{ denotes using } i \text{ units of m}}$$

$$= \frac{1}{(1 - x)^m}$$

Here in the final product of the above expression, the coefficient of say x^j, would denote the number of non-decreasing subsequences of j length.
So our answer would be sum of coefficients of $x^y, x^{y+1} ....x^n$, where $y = \lceil \frac{n}{m} \rceil$

Now coefficient of x^i in $(1 - x)^{-m}$ is:

$$\binom{m + i - 1}{i} = \binom{m + i - 1}{m - 1}$$

Thus our answer is

$$\binom{m + y - 1}{m - 1} + \binom{m + y}{m - 1} + ... + \binom{n + m - 1}{m - 1}$$

In order to solve this we use the following property of combinations:

$$\binom{A}{B} + \binom{A}{B + 1} = \binom{A + 1}{B + 1}$$

Now adding and subtracting $\binom{m+y-1}{m}$, we get

$$\binom{m+y-1}{m-1}+\binom{m+y-1}{m}+\binom{m+y}{m-1}+.....+\binom{n+m-1}{m-1}-\binom{m+y-1}{m}$$

Using property above repeatedly ,we get our final answer as:

$$\binom{m+n}{m}-\binom{m+y-1}{m}$$

TIME COMPLEXITY:

$O(log(N+M))$, for each test case.

PREREQUISITES:

Catalan Number, Dynamic Programing, Difference Array, Longest Increasing subsequence

PROBLEM:

This problem has two subtasks worth 50 *points each.*

You are given an integer N. You also have N empty sets $S_1, S_2, \ldots, S_N$.

Consider all the arrays A of length N such that $0 \leq A_i \leq 1$ for each $1 \leq i \leq N$ (there are 2^N such arrays in total).

For each of these 2^N arrays, we compute an array B of length N where $B_i =$ length of longest non-decreasing subsequence for the prefix of length i of array A. (For example: if $N = 5$ and $A = [0, 1, 0, 1, 1]$, then $B = [1, 2, 2, 3, 4]$.) Next we add the array B to the set S_{B_N} (For example, $B = [1, 2, 2, 3, 4]$ would be added to S_4).

Output the size of sets $S_1, S_2, \ldots, S_N$ as N space-separated integers. Since the size of the sets might be large, output the values modulo $10^9 + 7$.

Note that the S_i are *sets*, and hence do not contain duplicate elements. Please look at the sample cases below for an explained example.

EXPLANATION:

Difference array: D_i of a given array A is defined as $D_i = A_i - A_{i-1}$ (for $0 < i < N$) and $D_0 = A_0$ considering 0 based indexing.
Observation 1: For any array B (The array formed by computing the length of longest non-decreasing subsequence for the prefix of length i of array A) its difference array consists of only 0 and 1 as the length of longest non decreasing subsequence cannot increase by more than 1 in any case.
Observation 2: For any array B, each B_i is at least $ceil(i/2)$. Simply take the maximum of count of zeroes or the count of ones in the prefix.
Observation 3: In the difference array formed by any array B, For any prefix length, the number of ones is greater than or equal to the number of zeroes in it.
Since each B_i is greater than $ceil(i/2)$ and B_i is same as number of ones in the prefix of length i of difference array. Let C_1 be the count of ones and C_0 be the count of zeroes in the prefix of length i of difference array.
$\therefore C_1 \geq ceil(i/2)$ Let i be even then $C_1 \geq i/2 \implies C_1 >= (C_1 + C_2)/2 \implies (C_1 >= C_2)$. Similar analysis can be carried out when i is odd.
Observation 4: Every such difference array is valid. An example of the array which produces the same difference array is the difference array itself. Example Let difference array be $[1, 0, 1, 0]$ and let A be $[1, 0, 1, 0]$, Then array B is $[1, 1, 2, 2]$ difference array is $[1, 0, 1, 0]$. Each difference array corresponds to a different array B.
Thus we need to find difference arrays of size N which have total number of ones (same as B_N) in them as i for each $1 <= i <= N$

Dp solution
Let $dp[i][j]$ represent we are at index i of the array and there are j ones in the prefix of the array making sure there are no more zeroes than ones in the prefix.Initialize $dp[0][0]$ as 1.
Then either the element at i^{th} index is 1 or it is 0. Consider the dp states for both these cases as:
Then $dp[i][j] = dp[i-1][j-1] + (i - j \leq j)?dp[i-1][j] : 0$
The answer to the problem is $dp[n][i]$ for all i from 1 to N.
But this is and $O(N^2)$ solution and not enough to pass all the test cases.

Combinatorics Solution

Let ${}^{N}C_{K}$ denote the usual binomial coefficient, i.e. number of ways to select K objects from set of N objects). Then number of such difference arrays having i ones in total comes out to be ${}^{N}C_{N-i} - {}^{N}C_{N-i-1}$ if $2 * i > N$ else 0.

Explanation: Let $f(K, N)$ be the number of $B - arrays$ of length N such that the last element is K. We want to find $f(1, N), f(2, N), ..., f(N, N)$. $f(K, N) = 0$ when $K < N/2$ or when $K > N$, and $f(0, 0) = 1$. We have the recurrence $f(K, N) = f(K - 1, N - 1) + f(K, N - 1)$. Think of this as lattice paths on the xy-plane. We start at $(0, 0)$ and reach (K, N) using two types of moves:

1. Move one step up
2. Move diagonally up-right

The only restriction is that we aren't allowed to cross the line $y = 2 \cdot x$. Now let's bring this to a more well-known form. Suppose that, instead of moving diagonally up-right, we only moved right. What happens then? If we would normally reach (K, N), now we would reach $(K, N - K)$ instead because the number of times we moved right stays the same, and the number of times we moved up decreases by K. What about the restriction that we don't cross $y = 2x$? We can see that in the new model this translates to the restriction that we don't cross the line $y = x$

Proof: Consider a path that crosses the line $y = 2 \cdot x$ in the first model. Let the first time this happens be at the point $(s, 2s + 1)$ To reach $(s, 2s + 1)$, we must have made $2s + 1$ moves of which s were diagonal and $s + 1$ were up. In the new model, this translates to s right moves and $s + 1$ up moves, hence reaching $(s, s + 1)$ Similarly, reaching $(s, s + 1)$ in the new model means reaching $(s, 2s + 1)$ in the old model. So, all we need to do is count the number of standard lattice paths from $(0, 0)$ to $(K, N - K)$ that don't cross the line $y = x$. This can be calculated by standard reflection principle stuff. The total number of paths without any constraints is just $C(K + N - K, N - K) = C(N, N - K)$ via reflecting bad paths about $y = x + 1$, We can derive that the total number of bad paths equals the total number of unconstrained paths to $(N - K - 1, K + 1)$, which is $C(N - K - 1 + K + 1, K + 1)$ = $C(N, K + 1)$. The answer is then their difference

We're looking for the number of arrays of $0s$ and $1s$ such that $count(1) \geq count(0)$ for every prefix. Starting at $(0, 0)$, consider adding a 1 to the array as moving right and adding a 0 as moving up. Then, for length N and K ones, you reach the point $(K, N - K)$ without crossing the line $y = x$

Please refer to this article for refection principle: **Reflection principle**

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Factorisation, Small to Large

PROBLEM:

You are given a rooted tree T with N vertices. Each vertex v has an integer A_v associated with it. For each v, you need to output the number of distinct divisors of $\prod_{u \in \text{subtree of } v} A_u$.

EXPLANATION:

We can precompute the prime factorisation of all numbers between 1 and 10^6 in $10^6 \times \log 10^6$ steps, which is fast enough. We will assume we have these prime factorisations available.

Now, if $x = \prod_{i=1}^{k} p_i^{y_i}$ where p_i are prime and y_i are positive integers, then the number of distinct divisors of x is $\prod_{i=1}^{k}(y_i + 1)$.

Now, let's say we are given the prime factorisation (modulo $10^9 + 7$) and number of divisors d of a large number X. We want to compute the prime factorisation and number of divisors d' of $X \cdot x$, where $x \leq 10^6$. Let's say $X = \prod_{p \in P} p^{y_p}$, $x = \prod_{p \in P'} p^{z_p}$. Then $d' = d \cdot (\prod_{p \in P' \setminus P} z_p + 1) \cdot (\prod_{p \in P \cap P'} \frac{x_p + y_p + 1}{y_p + 1})$. The complete prime factorisation can be similarly computed. Note that there are at most P' algebraic operations here, and $P' \leq \log 10^6 < 20$. Therefore, we can do this fast.

Now our algorithm is easy. If v is a leaf, the answer can be computed directly using prime factorisation of A_v. For any non leaf node v, we can use small-to-large along with the merging algorithm we saw above to compute the answer fast.

TIME COMPLEXITY:

Roughly, the time complexity is $O(1e6 \log 1e6 + 20N \log^3 N)$.

PREREQUISITES:

Binary lifting

PROBLEM:

You are given a tree and Q queries on it. Each query is of the form (u, K), where you need to find the K-th vertex encountered if a DFS is started from u.
Queries must be answered online.

EXPLANATION:

There's an obvious $\mathrm{O}(N)$ solution for each query, where you perform the DFS directly. This is of course too slow, and there's no obvious way to optimize it.

Instead, let's do something else.
Let's root the tree at vertex 1 and start a DFS, thus computing the `tin` values for every vertex from 1. From now on, we assume the tree to be rooted at 1 whenever necessary, for example when talking about subtrees/children/ancestors/etc.

Let s_v denote the size of the subtree rooted at vertex v.

Now, let's see what happens when we have a query (u, K).
For convenience, let's say $c_1 < c_2 < \ldots < c_x < p < c_{x+1} < \ldots < c_r$, where the c_j are the children of u and p is its parent.
Then,

- If $K \leq 1 + s_{c_1} + s_{c_2} + \ldots + s_{c_x}$, then the answer to this query is simply the answer to the query $(1, K + tin[u])$ instead (which has been precomputed already).
- If $K > 1 + s_{c_1} + \ldots + s_{c_x} + (N - s_u)$, the answer to this query is the answer to query $(1, K + tin[u] - (N - s_u))$ (once again, precomputed).
- Otherwise, we have $1 + s_{c_1} + s_{c_2} + \ldots + s_{c_x} < K \leq 1 + s_{c_1} + \ldots + s_{c_x} + (N - s_u)$. In other words, the answer to this query doesn't lie inside the subtree of u, and we have to look outside.

To deal with the third case, one obvious way would be to simply move to the parent p of u, update K appropriately, and once again run this process.
Updating K can be done with a bit of casework, once again depending on the relative order of u as a child of p.

However, this can degenerate to $\mathrm{O}(N)$ per query, since you might have to move up to a parent $\mathrm{O}(N)$ times.

Notice that the only operation that really needs to be sped up is the 'move to parent' operation, where it'd be nice if we were able to move up multiple steps at the same time.

This is exactly what binary lifting accomplishes!

In fact, that's pretty much the remainder of the solution: use binary lifting to maintain appropriate data so that each query can be answered in $\mathrm{O}(\log N)$ time.

Unfortunately, the devil is in the details: the hard part here is maintaining 'appropriate data' across the lift.
A bunch of things need to be maintained so that a query can be answered properly, for example:

- Ancestors of elements
- Subtree sizes
- The order in which children of a vertex are visited, and the total number of other vertices visited before entering this one (from the perspective of the child).
- Left and right borders of vertices (note that the hard part above was dealing with K when it was in the 'middle' of the values for u. We maintain the left and right borders of this middle).

It all reduces to maintaining a few formulae in terms of these values, and then answering a query is a simple binary lift in $\mathrm{O}(\log N)$ time where u and K are changed appropriately.

I recommend looking at the code linked below for how to implement this, if you are stuck.

PREREQUISITES:

Bit Manipulation, Graph Traversal (DFS)

PROBLEM:

Chef has a tree consisting of N nodes, rooted at node 1. The parent of the i^{th} $(2 \leq i \leq N)$ node in the tree is the node P_i.

Chef wants to assign a binary value (`0` or `1`) to every node of the tree. The **Xor-value** of a node is defined as the **bitwise XOR** of all the binary values present in the **subtree** of that node.

Help Chef to assign values to the nodes of the tree in such a way that the sum of *Xor-value* of all the N nodes in the tree is **exactly** K.

It can be shown that such an assignment always exists. If there are multiple possible answers, you may print **any of them**.

EXPLANATION:

Hint

You can choose any K nodes to have Xor-value 1 and other $N-K$ nodes to have Xor-value 0.
We know that the Xor-value of any node is the Xor of all the Xor-values of its children and the value of the current node.

Xor-value of node = Value of node $\bigoplus$ Xor of Xor-values of all children

This is because, for any node, irrespective of the Xor-values of its children we can always assign 0 or 1 to the current node so that Xor-value of the current node is of our choice.

Value of node = Chosen Xor-value of node $\bigoplus$ Xor of Xor-values of all children

Solution

Let us use the standard DFS graph traversal algorithm and choose the K nodes having Xor-value 1 to be the K nodes having the least Post-visit numbers i.e. the time at which the DFS call for a node finishes.

By using DFS it is clear that when we are at a point to decide the value of a node, all the values of the nodes in its subtree are already decided. We also know the Xor-values of all its children and using this information we can always make the Xor-value of the current node of our choice.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

You are given $N (6 \leq N \leq 1000)$ non-negative integers. In one move, you can choose three of them, say A, B, C, and replace these three with $A \oplus B, A \oplus C, B \oplus C$.

Find a sequence of at most 11111 moves to make all N numbers 0.

EXPLANATION:

In a lot of constructive tasks like this, it's not immediately obvious how to proceed. Instead, start by making some observations and looking at simpler cases.

One useful observation is the following:

- If we pick $A = B = C$, then all three get replaced with 0.

This is nice, since our aim is to make everything 0.

Now, let's try looking at a special case: what if every integer was either 0 or 1?

Solution

One algorithm to solve this case is as follows:

- First, if there are at least 3 ones, we can use our earlier observation to make all 3 of them zeros.
- Repeatedly doing this leaves us with either $0, 1$, or 2 ones.
  - If there are 0 ones, we're done, since everything is 0.
  - If there is one 1, our only non-trivial move is to pick $\{0, 0, 1\}$, which gives us $\{0, 1, 1\}$, i.e, we are now at the case with 2 ones. All that remains is to solve that case.
  - If there are two 1's, we can use the fact that $N \geq 6$ to ensure that there are also at least two zeros.
    - First, pick $\{0, 0, 1\}$, which gives us $\{0, 1, 1\}$.
    - Now notice that we have three ones, so pick those three and we're done.

Now that we have a solution to the simpler case of 0's and 1's, extending it to the general case isn't too hard: simply apply this solution independently to each bit!

That is, first apply this solution to make the 0-th bit of all the numbers 0.
Then, make the first bit of everything 0
Then, make the second bit 0, and so on.

Since $A_i \leq 10^9$, this only needs to be done 30 times. What about the number of moves?

Answer

Let's look at the number of moves for a single bit, then multiply this by 30.

In the worst case, we make $N/3$ moves of 3 ones, followed by 3 extra moves for when there's exactly one 1 left. This gives us a worst-case of $1000/3 + 3$ moves, let's say 340 is an upper bound.

Multiplying this by 30 gets us to 10200, which is well within the limit.

Solving for a single bit can easily be done in $O(N)$ time, giving us a $O(30 \cdot N)$ solution overall.

TIME COMPLEXITY

$O(30 \cdot N)$ per test case.

PROBLEM:

You are given two integers A and B.

Determine whether it is possible to construct two distinct binary palindromes S and T where both S and T have exactly A 0's and B 1's.

EXPLANATION:

If both A and B are even then the answer is “Yes”, because you can construct a string with $\frac{A}{2}$ 0's, followed by B 1's, followed by $\frac{A}{2}$ 0's.

Eg. if A = 6, B = 4, then you can construct 0001111000

You can construct the other string with $\frac{B}{2}$ 1's, followed by A 0's, followed by $\frac{B}{2}$ 1's.

Now, if both A and B are odd, then no palindrome exists, so the answer is “No”. This fact is easy to verify.

If exactly one of them is odd and also > 1, then the answer is “Yes” again. The construction for this case is similar to the first case. If you assume that A is odd and B is even. You can construct the first string with $\frac{A-1}{2}$ 0 's, followed by $\frac{B}{2}$ 1's, followed by a 0, followed by $\frac{B}{2}$ 1's, followed by $\frac{A-1}{2}$ 0's. The second string is $\frac{B}{2}$ 1's, followed by A 0's, followed by $\frac{B}{2}$ 1's.

If either A or B is 1, then the answer is “No”.

TIME COMPLEXITY:

Time complexity is $O(1)$.

PREREQUISITES

Two pointers

PROBLEM

You are given a string S of length N. Start traversing the string from the left end. Everytime you encounter a vowel, **reverse** the entire substring that came before the vowel.

Print the final string after traversing from left to right and performing all the operations.

EXPLANATION

- The brute force solution to this is to traverse the string and reverse the entire substring before any vowel. Each reversal takes $O(N)$ time, which would make the worst-case complexity of this approach $O(N^2)$.
- A more efficient way is to iterate from the right end of the string, and use 2-pointers technique.

Let's say that the initial string is S and our final string is going to be T.

A useful observation is that any character $S[i]$ will never be reversed more times than $S[i-1]$, that is, no character is reversed more times than the preceding character. In particular, the last character $S[N-1]$ will stay at the same position, i.e. $T[N-1] = S[N-1]$.

Let's iterate from the right end of the string. This way, we can keep account of the number of vowels that have been encountered so far, as this will equal the number of reversals that need to be performed. Let the number of vowels encountered be V. What matters to us is the **parity** of V. If we encounter an even number of vowels, then there will be an even number of reversals of the current character and it should end towards the right side of the string. Similarly, if we encounter an odd number of vowels, the current character should end towards the left side of the string.

Note that iterating from the right end is the key since, the position of a character $S[i]$ is only determined by the characters following it. If we change any of the letters in the substring $S[1]$ to $S[i-1]$, it will not affect the final positions of the characters $S[i]$ to $S[N-1]$.

Explanation of sample test case

Consider the sample from the problem. The initial string $S = abcdefghij$. Let's start from the right end. Initially, we encounter j and place it to the right end of T. Next up, we encounter i. Since i is a vowel but it itself won't be reversed, we place it to the right end of T. Now the final 2 letters of T have been determined, they are ij. Moreover, the number of vowels, $V = 1$.

Now, the next letter is h. Since $V = 1$, there will be one reversal and h will go to the front of the string. Similarly, the next two letters, f and g will go towards the front of the string. The string T now statrts with hgf and ends with ij. The middle letters are yet to be determined.

Next, we encounter e, which again goes to the front of the string, but now V is incremented and becomes 2. After this d, c, b, a will go to the back of the string, as they will be reversed twice. The final string $T = hgfeabcdij$.

TIME COMPLEXITY

The time complexity is $O(N)$.

PREREQUISITES:

Depth first search, Dynamic programming, Trees

PROBLEM:

Arun has a rooted tree of N vertices rooted at vertex 1. Each vertex can either be coloured black or white.

Initially, the vertices are coloured $A_1, A_2, \ldots A_N$, where $A_i \in \{0, 1\}$ denotes the colour of the i-th vertex (here 0 represents white and 1 represents black). He wants to perform some operations to change the colouring of the vertices to $B_1, B_2, \ldots B_N$ respectively.

Arun can perform the following operation any number of times. In one operation, he can choose any subtree and either paint all its vertices white or all its vertices black.

Help Arun find the minimum number of operations required to change the colouring of the vertices to $B_1, B_2, \ldots B_N$ respectively.

EXPLANATION:

This problem can be solved using **dynamic programming** as it has optimal substructure and overlapping subproblems.
The tree is rooted at node 1.
Let $black_u$ represent the minimum number of steps needed to make $A_i = B_i$ for each node i in the subtree of node u when the complete subtree of node u is painted black with an operation.
Let $white_u$ represent the minimum number of steps needed to make $A_i = B_i$ for each node i in the subtree of node u when the complete subtree of node u is painted white with an operation.
Let $none_u$ represent the minimum number of steps needed to make $A_i = B_i$ for each node i in the subtree of node u without doing any operation on u.
Initialize each of the value $black_u, white_u \; and \; none_u$ for all u where $1 \leq u \leq N$ with 0

Start the dfs at node 1. Let us suppose we are at some node u during the dfs traversal then for this node we have three values :
$none_u += ((A_u != B_u)?INF : min(1 + black_x, 1 + white_x, none_x)$ over all children x of node u
We can choose to colour any child's subtree either black, white or leave the child untouched whichever gives minimum number of operations we add it to $none_u$.
$black_u += ((!B_u)?1 + \sum white_x : \sum black_x)$ over all children x of node u
If in the end we need the colour of node u as white we need to first colour the whole subtree of u white otherwise we can just leave the node u untouched and calculate the answer for $black_x$ for all children x of node u and add them to $black_u$.
$white_u += ((B_u)?1 + \sum black_x : \sum white_x)$ over all children x of node u
If in the end we need the colour of node u as black we need to first colour the whole subtree of u black otherwise we can just leave the node u untouched and calculate the answer for $white_x$ for all children x of node u and add them to $white_u$.
The answer to the problem is $min(none_1, 1 + black_1, 1 + white_1)$;

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

(Optional) Dynamic programming

PROBLEM:

You have a permutation P of $\{1, 2, \ldots, N\}$. In one move, you can negate any of its elements.
A subarray is called *weird* if it can be sorted in ascending order by applying this move several times. Count the number of weird subarrays in P.

EXPLANATION:

In tasks like this which require you to count objects satisfying a certain property, it's useful to get a simpler criterion of that property.

Let's look at a weird array $A = [A_1, A_2, \ldots, A_k]$. What does this tell us about the elements of A?
One obvious fact is that in the final array, all the negative elements *must* come before all the positive elements — otherwise, there's no way for the final array to be sorted.
So, there is some index p such that $A_1, A_2, \ldots, A_p$ are negated, and $A_{p+1}, \ldots, A_k$ are not. Looking at each of those parts individually,

- We want $-A_1 < -A_2 < \ldots < -A_p$, i.e, $A_1 > A_2 > \ldots > A_p$
- We also want $A_{p+1} < A_{p+2} < \ldots < A_k$

In other words, A must look like a 'valley': first decreasing, then increasing (purely increasing/decreasing is also ok). It's also obvious that any such array is *weird*, since it can be sorted by negating the first part.

So, we just need to count the number of valleys in P. There are several ways to do this, a couple will be mentioned below:

- Let's call an index i a *hill* if $P_{i-1} < P_i > P_{i+1}$. Note that a subarray is a valley if and only if it doesn't contain a hill index along with both of its neighbors.
  - Suppose we find all hill indices. Then, we can simply count the number of weird indices between each adjacent pair of them to obtain our answer.
  - Counting the number of subarrays between two hills is simple combinatorics.
  - You may refer to the setter's code (below) for this approach.
- Another method is to directly count valleys.
  - Suppose we fix the deepest point of the valley to be index i. Let's count the number of valleys like this.
  - The left of i should be some decreasing sequence, and the right should be increasing.
  - Let $left_i$ denote the longest decreasing sequence ending at i, and $right_i$ denote the longest increasing sequence starting at i. These two can be found with dynamic programming in $\mathrm{O}(N)$.
  - Then, we add $left_i \times right_i$ to the answer, to account for all choices of subarrays with i as the deepest point (also including purely increasing/decreasing subarrays)
  - Adding up $left_i \times right_i$ across all $1 \leq i \leq N$ gives us the answer
  - You may refer to the editorialist's code (below) for this approach.

TIME COMPLEXITY

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

PROBLEM:

Given an integer N, construct a simple connected undirected graph on N vertices such that no two vertices of the same degree have a prime difference.
Among all such graphs, minimize the maximum degree.

EXPLANATION:

This can be thought of as a coloring problem: all vertices with the same degree share the same color, and vertices with different degrees have different colors.
We'd like to ensure that no two vertices with the same color have a prime difference; while minimizing the number of colors used.

Let's leave aside lower values of N for now, and consider $N \geq 8$.

When $N \geq 8$, we must use at least 4 colors.

Proof

This is easy to prove: all four of $1, 3, 6, 8$ must have different colors, because the difference of any pair is a prime.

In fact, when $N \geq 8$ it's always possible to construct a graph with maximum degree 4, so this is also optimal.
One construction is as follows:

Construction

First, let's solve for $N \geq 12$.

Let's assume $N = 4k$ for some $k \geq 3$. Color elements based on their values modulo 4: this will ensure that no same-colored pair has a prime difference.
From here on, the elements will be referred to based on their colors: $1, 2, 3, 4$.

- Join the 4's and 2's together in an alternating cycle of size $2k$
- Join the 3's together in a separate cycle of size k
- Finally, join one 1 and one 3 to each 4.

It's easy to see that all degrees are satisfied this way.
Now, for $N = 4k + r$ where $r > 0$:

- If $r = 1$, treat the extra element as a 2 and put it in one of the cycles created
- If $r = 2$, treat the extra elements as a 1 and a 3. Put the 3 in one of the cycles and join the 1 to it.
- If $r = 3$, treat the extra elements as $1, 2, 3$ and do both of the above.

Of course, when doing this make sure to relabel the colors so that the difference of 4 invariant is maintained.
This solves everything ≥ 12.

For $N = 8$, create two cycles as $(4, 4, 2)$ and $(3, 3, 2)$. Then join the 1's and the 3's to the 4's as our initial construction.
$N = 9, 10, 11$ can use the same modification for $4k + r$ as above.

This leaves us only $N \leq 7$ to deal with. This can be done in several ways: for example, you could work out the answer by hand, or bruteforce to find a solution.
The answers for $N = 2, 3, 4$ are already in the samples, so only $5, 6, 7$ need to be solved.

TIME COMPLEXITY

$O(N)$ per test case.

PREREQUISITES:

Dynamic programming

PROBLEM:

You have two arrays A and B of length N. For each K from 0 to $N-1$, find the following:

- Let C be an array with $0 \le C_i \le B_i$
- At least K elements of C must be 0
- $sum(A) = sum(C)$
- Under these constraints, find the minimum value of $\sum_{i=1}^{N} |A_i - C_i|$.

EXPLANATION:

Let's fix a value of K and see what happens to C.

There are at most $N-K$ non-zero indices; let S denote the set of these non-zero indices. Then, we have the following:

- Let $B_S = \sum_{i \in S} B_i$. We must have $B_S \ge sum(A)$, otherwise it's impossible for $sum(C) = sum(A)$ to hold. For now, let's assume this condition holds.
- We can assign C using the following strategy:
  - If $i \notin S$, assign $C_i := 0$.
  - If $i \in S$ and $B_i \le A_i$, assign $C_i := B_i$. This is the best we can do at this index. Such indices contribute a value of $A_i - B_i$ to the result.
  - If $i \in S$ and $B_i > A_i$, assign $C_i := A_i$ initially.
  - Now, we need to add some values to indices of the third type to ensure $sum(C) = sum(A)$. Note that it doesn't really matter which index a value is added to: adding x to any such index increases the result by x.
  - Since $B_S \ge sum(A)$, it's always possible to achieve $sum(C) = sum(A)$, so we only need to find out how much is added — that will tell us the result.
- Finding out how much is added is simple: we have to add A_i for each $i \notin S$, and then $A_i - B_i$ for each $i \in S$ with $A_i \ge B_i$.

So, for the S we chose, the result can be computed as follows:

- Let B_S be as denoted above. Similarly, let $A_S = \sum_{i \in S} A_i$.
- Further, let $D = \sum_{i \in S} \max(0, A_i - B_i)$.
- Then,
  - The total contribution of indices $i \in S$ with $B_i \le A_i$ is D
  - The total contribution of indices $i \in S$ with $B_i > A_i$ is $sum(A) - A_S + D$
  - The total contribution of indices $i \notin S$ is $sum(A) - A_S$
- Adding all these together, the result for this set S is $2 \cdot (D + sum(A) - A_S)$

We would like to choose S in such a way that this is minimized. $sum(A)$ is a constant, so this is equivalent to minimizing $D - A_S$.
Note that a given index contributes $(\max(0, A_i - B_i) - A_i)$ to $D - A_S$ if it is chosen in S.

We also need to keep track of B_S, since we need to ensure it is greater than $sum(A)$.

This allows us to write the following knapsack-style dynamic programming solution:

- Let $dp_{i,k,x}$ denote the minimum possible value of $D - A_S$ for a set of size k chosen from indices $1, 2, \ldots, i$, whose sum of B_i values is x.
- At position i, we have two choices: either take index i into S, or don't
  - If i is not taken, we simply have $dp_{i-1,k,x}$
  - If i is taken, we have $dp_{i-1,k-1,x-B_i} + \max(0, A_i - B_i) - A_i$
  - $dp_{i,k,x}$ is the minimum of these two values.

This gives us a solution in $\mathrm{O}(N^4)$ with a pretty low constant factor (more specifically, it is $\mathrm{O}(N^2 \cdot sum(B))$, but $sum(B) \leq 10^4$ anyway).

However, it currently uses $\mathrm{O}(N^4)$ memory, which might be too much. Note that by iterating from left to right, when we are at i, we only care about the dp values at $i - 1$. This allows us to cut down the memory to $\mathrm{O}(N^3)$.

TIME COMPLEXITY

$\mathrm{O}(N^4)$ per test case.

PROBLEM:

Chef can climb either Y stairs or 1 stair in a single move. What's the minimum number of moves to reach stair X ?

EXPLANATION:

If possible, it's better to use one move to climb Y stairs rather than 1, since $Y \geq 1$.

That gives an easy greedy strategy.

- Let C denote the current stair Chef is on.
- If $C + Y \leq X$, use one move to climb Y stairs.
- Otherwise, climb one stair.

Directly implementing this using a loop is enough to get AC.

Thinking a little more should also give you a simple formula for the answer:

$$\lfloor \frac{X}{Y} \rfloor + (X \bmod Y)$$

where $(X \bmod Y)$ is the remainder when X is divided by Y, represented by `X % Y` in most languages.

TIME COMPLEXITY

$O(1)$ per test case.

PREREQUISITES:

Binary search or 2-pointers

PROBLEM:

You have an array A of distinct elements. You can choose an integer X and set $A_i \leftarrow A_i \oplus X$ for every $1 \leq i \leq N$.

If X is chosen optimally, what is the maximum possible length of the longest increasing subarray of the final array?

EXPLANATION:

We want the longest increasing subarray, so let's look at some specific subarray and see where that gets us.
Suppose we want to make the subarray $[A_L, A_{L+1}, \ldots, A_R]$ increasing. Then, it's enough to choose X so that $A_L < A_{L+1}, A_{L+1} < A_{L+2}, \ldots, A_{R-1} < A_R$.
That is, we only really care about adjacent elements: making them increasing automatically makes the subarray as a whole increasing.

So, let's look at adjacent elements A_i and A_{i+1}. Suppose we make $A_i < A_{i+1}$ after xor-ing with X. What sort of restrictions would this put on X?

Answer

Note that only the highest differing bit between A_i and A_{i+1} matters, since this is what determines when one binary number is larger than another.
So, let k be the highest differing bit between A_i and A_{i+1} (for example, the highest differing bit between $4 = 100_2$ and $5 = 101_2$ is 0 ($2^0 = 1$), and the highest differing bit between $4 = 100_2$ and $7 = 111_2$ is 1 ($2^1 = 2$)).

- If $A_i < A_{i+1}$ originally, then the k-th bit is set in A_{i+1} and not in A_i. Any X we choose *must* have the k-th bit **unset** to preserve this inequality.
- If $A_i > A_{i+1}$ originally, then the k-th bit is set in A_i and not in A_{i+1}. Any X we choose *must* have the k-th bit **set** to reverse this inequality.

The other bits in X can be anything, only this bit has a restriction.

So, let's compute an array B of length $N-1$, where B_i denotes the type of restriction on X needed for $A_i < A_{i+1}$ to hold. Computing B is easy to do: simply iterate across bits in descending order and find out when A_i and A_{i+1} differ.

Now, note that making an array $[L, R]$ increasing is equivalent to saying that we consider all the restrictions $B_L, B_{L+1}, \ldots, B_R$, and there are no conflicts between them. A conflict here is a situation where, for some bit k, the restrictions tell us that it must be both on and off: this is clearly impossible.

So, let's fix the left endpoint L. Then, let's find the largest possible R such that there is no conflict between relations $B_L, \ldots, B_{R-1}$. The largest possible increasing subarray starting at L is then $[L, R]$, with length $R - L + 1$.
Doing this for every L and taking the maximum length across all of them will give us our answer.
However, this is an $O(N^2)$ algorithm: how do we improve it?

To speed this up, we notice that the optimal right endpoints form a non-decreasing function.
That is, if R_1 and R_2 are the optimal endpoints for L_1 and L_2 respectively, and $L_1 \leq L_2$, then $R_1 \leq R_2$.
This is easy to prove: simply note that if $[L, R]$ has no conflicts, then $[L+1, R]$ also has no conflicts.

This tells us that our algorithm can be optimized using a 2-pointer approach, as follows:

- Start at $L = 1$ and $R = 1$.
- Maintain a set S of the currently active restrictions.
- Then, repeat the following process:
  - While $[L, R + 1]$ has no conflicts, increase R by 1.
    - This is done by checking if the opposite restriction of B_R is currently in S or not.
    - If R is increased, add B_R to S.
  - Now we have the optimal R for this L. Set $ans = \max(ans, R - L + 1)$.
  - Now, increase L by 1 but don't change R. This is done by simply removing B_L from S.

S can be easily maintained using `std::multiset` or a similar data structure; it's even possible to just use an array since the restrictions themselves are of small values.

TIME COMPLEXITY

$O(N \log N)$ per test case.

PREREQUISITES:

Bit Manipulation

PROBLEM:

JJ has three integers N, A and B where $0 \leq A, B < 2^N$. He wants to find a third integer X such that:

- $0 \leq X < 2^N$
- the value of $(A \oplus X) \times (B \oplus X)$ is maximum.

Can you help him find such an integer X? If there are multiple integers which satisfy the given conditions, print any.

EXPLANATION:

We need to consider only the first N bits of A, B, and X because all are $< 2^N$. Let us consider the i^{th} bit of A and B. There are 3 possibilities:

- i^{th} bits of both A and B are set (1). If we keep the i^{th} bit of X as 0 then the i^{th} bits in both $(A \oplus X)$ and $(B \oplus X)$ are set. This clearly maximizes the product.
- i^{th} bits of both A and B are not set (0). If we keep the i^{th} bit of X as 1 then the i^{th} bits in both $(A \oplus X)$ and $(B \oplus X)$ are set (1). This clearly maximizes the product.
- i^{th} bits of A and B are both different i.e. one set (1) and one not set (0). In this case irrespective of what the i^{th} bit in X is, one among the i^{th} bits in $(A \oplus X)$ and $(B \oplus X)$ is set (1) and the other is not set (0).

From the first two points it is clear that we can write $(A \oplus X)$ as $C + D$ and $(B \oplus X)$ as $C + E$, such that C is the number having set (1) bits at all positions where the bits of A and B are same and has all other bits not set (0).
From the third point, D and E have a fixed sum $(2^N - C - 1)$ which is equal to that number having set (1) bits at all positions where the bits of A and B are different and has all other bits not set (0).

The product to be maximized simplifies to $(C + D) \times (C + E) = C^2 + C \times (2^N - C - 1) + E \times D$. The first two terms are fixed based on A, B, and N. The last term involves maximizing the product of two integers having fixed sum. For this, it is clear that we will choose X so that D and E are as close as possible.

Let us say that for D the most significant bit where A and B differ is set (1), then E should be such that it has set bits at all other positions (leaving the most significant position) where A and B differ. This is because $2^K \geq 2^{K_1} + 2^{K_2} + ...2^{K_X}$, where all $K_1, K_1, \ldots K_X$ are distinct and $< K$.

In summary, X is chosen so that:

- $(A \oplus X)$ and $(B \oplus X)$ have set bits at all positions where A and B have same bits.
- Either $(A \oplus X)$ or $(B \oplus X)$ has a set bit at the most significant bit position where A and B differ and at all other positions it has 0 bits and vice versa.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Bitwise XOR, C++ Map

PROBLEM:

Chef has two arrays A and B, each of length N.

A pair (i,j) $(1 \leq i < j \leq N)$ is said to be a good pair if and only if
$A_i \oplus A_j = B_i \oplus B_j$

Here, $\oplus$ denotes the bitwise XOR operation.

Determine the number of good pairs.

EXPLANATION:

Let us simplify the equation given in the problem by taking XOR with $A_j \oplus B_i$ on both sides.
$A_i \oplus A_j \oplus A_j \oplus B_i = B_i \oplus B_j \oplus A_j \oplus B_i$
$A_i \oplus B_i = A_j \oplus B_j$
$\therefore$ We need to find the number of pair (i,j) $(1 \leq i < j \leq N)$ such that $A_i \oplus B_i = A_j \oplus B_j$
Iterate on the arrays from $i = 1$ to N, keeping track of $A_i \oplus B_i$ with the help of a map data structure. For each index i add the count of $A_i \oplus B_i$ to the answer then update the count of $A_i \oplus B_i$ in the map.

TIME COMPLEXITY:

$O(Nlog(N))$ or for each test case.