PREREQUISITES

Observation, Fenwick Tree

PROBLEM

Given an array A with N distinct integers, a subarray $[L, R]$ is called Tsetso subarray if $A_L \cdot A_R = max(A_L, A_{L+1} \dots A_{R-1}, A_R)$

Answer queries of the form. Given interval $[L, R]$, count the number of Tsetso subarrays completely inside the query range.

QUICK EXPLANATION

- The number of Tsetso subarrays is approximately $N * log_2(N)$.
- For element A_p, consider all intervals $[l, r]$ such that $l \leq p \leq r$ and A_p is the maximum element in subarray $[l, r]$. A_l and A_r must be factor of A_p in order to get a Tsetso subarray.
- Iterate over elements $l \leq le \leq p$ on one side (say left), then we can check if A_p/A_{le} is present in range $[p, r]$. If it is present at position ri such that $p \leq ri \leq r$, then subarray $[le, ri]$ is a Tsetso subarray.
- After finding all subarrays, we just need to count the number of subarrays completely inside each query ranges, which can be done by sorting intervals by the right end and answering queries offline.

EXPLANATION

Since all elements are distinct, we can see that in an interval, no two elements are maximum at the same time. Let's say the maximum of the array is present at position p. Then all subarrays $[L, R]$ having $L \leq p \leq R$ shall have it's maximum value A_p. So all Tsetso subarrays would have A_L and A_R as factors of A_p. This, combined with the fact that all elements in the array are distinct, would suggest that number of subarrays would be quite less than N^2.

Let's try finding all Tsetso subarrays.

Finding all Tsetso subarrays

Considering subarray $[L, R]$ where A_p is the maximum in range $[L, R]$. Then all subarrays $L \leq p \leq R$ have A_p as maximum. If we fix A_L, $A_R = A_p/A_L$ must hold. Hence, let's iterate i from L to p. We know the value $x = A_p/A_i$. If value x is present at position j where $p \leq q \leq R$, then subarray $[i, j]$ is a Tsetso subarray.

Hence, we can find all Tsetso subarrays with A_p as maximum. Now we only need to consider subarrays in the range $[L, p-1]$ and $[p+1, R]$, which are similar sub-problems.

Since we needed to do $O(N)$ work (Consider ascending or descending array A) and we got two smaller sub-problems, this can lead to $O(N^2)$ time in the worst case to find all subarrays. But we can improve

Number of subarrays

Earlier, we iterated on one side. Now, let's choose to iterate on the smaller side. For example, if we are considering range $[3, 10]$ and we have $p = 8$, then let's iterate on the right side fixing A_R and use map/binary search to check if A_p/A_R is present in the range $[L, p]$.

While this may seem a small thing, it actually reduces time complexity to generate all subarrays from $O(N^2)$ to $O(N * log(N))$.

Let's say we start at interval $[L, R]$ and the maximum element is at position p. We do only $min(R-p, p-L)$ iterations, and got two subproblems $[L, p-1]$ and $[p+1, R]$. The worst-case happens when p is roughly in middle, leading to $(R-L)/2 iterations and two subproblems of size $N/2$.

We can write the worst case recurrence as $T(N) = 2 * T(N/2) + O(N)$ which leads to $O(N * log(N))$ time.

Also, since in each iteration one subarray is being considered, the number of Tsetso subarrays is roughly $N * log_2(N)$.

We can use RMQ, or Segment Tree or even sorting to process elements in decreasing order and finding for each element A_p, the range $[L, R]$ where A_p is maximum and finding all Tsetso subarrays.

Answer queries

Now we have all Tsetso subarrays in a single list. We need to answer the number of subarrays present completely inside the query range. This is somewhat well known, but I'd reiterate in brief.

Let's sort all the arrays, and all the queries by the right end in non-decreasing order. Before answering q-th query with range $[L, R]$, add all subarrays with right end $\leq R$ into a DS. Now, we only need to count the number of subarrays present in DS, which have left end $\geq L$.

We can see that subarrays having the right end beyond R wouldn't be added to DS, and the left end is taken care of by counting only numbers having the left end $\geq L$.

One popular choice for DS would be Fenwick Tree, where when adding subarray $[l, r]$ to DS, we increment position l by 1. When answering a query for the range $[L, R]$, we can simply query the sum of range $[L, R]$.

Since each subarray is added to DS only once, and all queries need to be processed only once, this solution works in $O(N * log^2(N) + Q * log(N))$ (Sorting and Fenwick Tree operations).

PREREQUISITES:

Bitwise operations, std::map

PROBLEM:

Chef was impressed by an array A of N non-negative integers. But, he was asked to solve the following problem in order to get this array.

Given a non-negative integer k, find the number of pairs (i,j) $(1 \leq i < j \leq N)$ such that the following condition holds true:
$(A_i \mid A_j) + (A_i \oplus A_j) + (A_i \,\&\, A_j) = k + A_j$, where

- $(A_i|A_j)$ denotes Bitwise OR operation,
- $(A_i \oplus A_j)$ denotes Bitwise XOR operation,
- $(A_i \,\&\, A_j)$ denotes Bitwise AND operation.

You, being Chef's friend, help him get this array by solving the above problem.

EXPLANATION:

Observation

$(a|b) = (a \oplus b) + (a \,\&\, b)$.
$(a + b) = (a \oplus b) + 2 \cdot (a \,\&\, b)$
$\therefore (a|b) + (a \oplus b) + (a \,\&\, b) = (a \oplus b) + (a \oplus b) + (a \,\&\, b) + (a \,\&\, b) = (a \oplus b) + (a + b)$.

The equation in the question is same as $(A_i \oplus A_j) + (A_i + A_j) = (K + A_j)$.
$\implies (A_i \oplus A_j) = (K - A_i)$.
$\implies A_j = ((K - A_i) \oplus A_i)$.
Thus for each index i we need to find indices j such that $i < j$ and $A_j = ((K - A_i) \oplus A_i)$. This can easily be done using a map.

TIME COMPLEXITY:

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

PROBLEM:

There are N students in a class. Recently, an exam on Advanced Algorithms was conducted with maximum score M and minimum score 0. The average score of the class was found out to be **exactly** X.

Given that a student having score **strictly greater** than the average receives an `A` grade, find the **maximum** number of students that can receive an `A` grade.

EXPLANATION:

Let us look at the two cases for this problem:

- $X = M$: Here it implies that every single student has scored M marks and since no student can score more than M marks. Thus in this case no student would receive A grade.
- $X < M$: Let us take a variable sum as the sum of marks of all students. Then

$$sum = N \times X$$

Now to maximise students who score more than average marks we assign $(X + 1)$ marks to as many students as possible, which would be our answer.

$$answer = \lfloor \frac{sum}{X + 1} \rfloor$$

TIME COMPLEXITY:

$O(1)$, for each test case.

PROBLEM:

Given two positive integers a and b. You have a number x which is initially 0. At first, you can add a to x any number of times. After that, you can divide x by b any number of times.

Determine if it is possible to make x equal 1.

EXPLANATION:

Say it is possible to make x equal to 1 by:

- first adding a to it n times, and then
- dividing it b, m times.

Mathematically, this can be written as

$$\frac{n \times a}{b^m} = 1 \implies n = \frac{b^m}{a}$$

Since n is an integer, the equation implies that a divides b^m for some m. This is true if and only if b is divisible by every prime factor of a.

Proof

If there exists some p such that $p|a$, then

$$p \nmid b \implies p \nmid b^m \implies a \nmid b^m$$

Thus, b should be divisible by every prime factor of a.

When m is large enough, the highest power of p that divides b^m *will exceed* the highest power of p that divides a, for all prime p, implying $a|b^m$.

All that remains now is to check if b is divisible by each prime factor of a.

Naive solution

We can calculate all prime factors of a in $\sqrt{a}$, using the **Sieve of Eratosthenes**. However, the time complexity over all test cases will be

$$O(T\sqrt{a})$$

which will TLE for the given constraints.

Let $g = \gcd(a, b)$. While $g \neq 1$, divide a by g and repeat. If at the end, a equals 1, then b is divisible by each prime factor of a, and not otherwise.

Proof

If there exists some p such that $p|a$ and $p \nmid b$, then $p \nmid \gcd(a, b)$, a will always have a factor of p and so will never become equal to 1.

When $p|b$, it can be seen that the highest power of p that divides a is successively reduced to 0 on dividing by g, which implies a becomes 1 eventually.

The time complexity of this approach can be found in the corresponding section below.

TIME COMPLEXITY:

Computing the gcd takes $O(\log \min(a, b))$. The number of prime factors in the factorisation of a is $\leq \log a$ (**why?**). Since in each division by g, the number of prime factors in the factorization of a is reduced (until g or a equals 1), the total time complexity is

$$O(\log \min(a, b) \times \log a)$$

PROBLEM:

CodeChef admins went on shopping at a shopping mall.

There are N shops in the mall where the i^{th} shop has a capacity of A_i people. In other words, at any point in time, there can be **at most** A_i number of people in the i^{th} shop.

There are X admins. Each admin wants to visit each of the N shops **exactly once**. It is known that an admin takes exactly **one** hour for shopping at any particular shop. Find the **minimum** time (in hours) in which all the admins can complete their shopping.

Note:

1. An admin can visit the shops in any order.
2. It is possible that at some point in time, an admin sits idle and does not visit any shop while others are shopping

EXPLANATION:

For each test case, we are given the number of shops N, the number of admins X and the **maximum** people that can shop at a particular store simultaneously.

Since each Admin **must necessarily visit all the** N **shops** once, so clearly the minimum time required in case of no restrictions is N.

When the maximum number of admins are limited for the shops, we first find the shop which allows for the **minimum** number of admins at the same time. This shop will tell us about the minimum time *in case the shifts **are more than*** N otherwise the answer will remain N.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Prefix and suffix sums

PROBLEM:

The alternating sum of an array A is $A_1 - A_2 + A_3 - \ldots + (-1)^{N-1}A_N$. At most once, you can take an odd-sized subarray of A and move it to the end. What is the maximum possible alternating sum the final array can have?

EXPLANATION:

There are $\mathrm{O}(N^2)$ odd-sized subarrays, so under the given constraints calculating the alternating sum when each one of them is moved to the end will be too slow.

Let's analyze how the alternating sum changes when we move subarray $[L, R]$ to the end. The array is now $[A_1, A_2, \ldots, A_{L-1}, A_{R+1}, \ldots, A_N, A_L, \ldots, A_R]$.
Note that the array is effectively split into three parts - the prefix $[1, L-1]$, the subarray $[L, R]$, and the suffix $[R+1, N]$.
Let's look at the contribution of each of those parts to the alternating sum separately.

The prefix

The elements of the prefix don't change their position whatsoever, and so the alternating sum of this part is $A_1 - A_2 + A_3 - \ldots + (-1)^{L-2}A_{L-1}$ — exactly the same as its contribution to the alternating sum in the original array.

The suffix

The elements of the suffix $[R+1, N]$ are moved such that A_{R+1} is at position L, A_{R+2} is at position $L+1$, and so on.
So, the alternating sum of this part is $(-1)^{L-1}A_{R+1} + (-1)^{L}A_{R+2} + \ldots$.
However, note that the length of the subarray is odd - i.e, $R - L + 1$ is odd, which means that L and R have the same parity.
Thus, the above sum can also be written as $(-1)^{R-1}A_{R+1} + (-1)^{R}A_{R+2} + \ldots + (-1)^{N-2}A_N$.

However, alternating sum in the original array is $(-1)^{R}A_{R+1} + (-1)^{R+1}A_{R+2} + \ldots$ — the sign of every element is flipped. Thus, this part contributes the negative of whatever it did to the original array.

The subarray

Let $k = R - L + 1$ be the length of the subarray.
The elements of the subarray are moved such that they now start from position $N - k + 1$ (because all the $N - k$ elements outside the subarray are placed before it).
The alternating sum of this part is then $(-1)^{N-k}A_L + (-1)^{N-k+1}A_{L+1} + \ldots (-1)^{N-k+R-L}A_R$.
Now,

- If $(-1)^{N-k} = (-1)^{L-1}$, the alternating sum of this part is exactly the same as its alternating sum in the original array, because all the signs will remain the same.
- If $(-1)^{N-k} \neq (-1)^{L-1}$, the alternating sum of this part is the negative of its alternating sum in the original array, because all its signs flip.

When do these cases happen?
We know that k is odd, so $N - k$ and $L - 1$ have the same parity if and only if N and L have the same parity — equivalently, N and R have the same parity (because L and R have the same parity).

Putting it together, we know the following:

- The alternating sum of the prefix stays the same.

- The alteranting sum of the suffix is the negative of its alternating sum in the original array.
- The alternating sum of the subarray is either the same or the negative, depending only on the parity of the endpoints of the subarray.

This gives us the following idea: suppose we fix the right endpoint R of the subarray. Let's try to find the optimal left endpoint L such that moving $[L, R]$ to the end gives us as large an alternating sum as possible. Computing this for every $1 \leq R \leq N$ will give us the answer.

Note that we are dealing with prefix and suffix alternating sums, so it is useful to precompute those.
Let

$$pref_i = A_1 - A_2 + A_3 - \ldots + (-1)^{i-1} A_i$$
$$suf_i = (-1)^{i-1} A_i + (-1)^i A_{i+1} + \ldots + (-1)^{N-1} A_N$$

Also note that once $pref_i$ has been computed, the alternating sum of any subarray $[L, R]$ can be obtained as $pref_R - pref_{L-1}$.

Now, fix the right endpoint R. There are two cases:

- Suppose R and N have the same parity. Then, for any $1 \leq L \leq R$ (where $R - L + 1$ is odd), the prefix $[1, L-1]$ and the subarray $[L, R]$ have the same alternating sum as the original array, while the suffix $[R+1, N]$ has its alternating sum negated.
  Putting this together, the total alternating sum is simply $pref_R - suf_{R+1}$, independent of choice of L.
- Suppose R and N have different parities. Then, for any $1 \leq L \leq R$ (where $R - L + 1$ is odd), the prefix $[1, L-1]$ has the same alternating sum, while the subarray $[L, R]$ and the suffix $[R+1, N]$ have their alternating sums negated.
  The total alternating sum is then $pref_{L-1} - suf_{R+1} - (pref_R - pref_{L-1})$, which equals $2 \cdot pref_{L-1} - pref_R - suf_{R+1}$.
  The latter two terms are independent of choice of L, so this expression is maximized when $pref_{L-1}$ is maximized.
  However, we already computed all the values of $pref_i$, so this is simple to do — just compute prefix maximums of the $pref$ array. Remember that L and R must have the same parity (which is different from the parity of N in this case), so be sure to compute the maximums only over those indices whose parity matches the parity of N.

Once the prefix sums, suffix sums, and prefix maximums are computed, the answer for a given R can be found in $\mathrm{O}(1)$. Thus, the algorithm as a whole runs in $\mathrm{O}(N)$.

TIME COMPLEXITY:

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

PREREQUISITES:

SOS-DP

PROBLEM:

Chef has a (0-indexed) **binary** string S of length N such that N is a **power of** 2.

Chef wants to find the number of pairs (i,j) such that:

- $0 \leq i,j < N$
- $S_{i|j} = S_{i\&j}$

(Here $|$ denotes the **bitwise OR operation** and $\&$ denotes the **bitwise AND operation**)

Can you help Chef to do so?

QUICK EXPLANATION:

What if we fix i?
Let S_1 be the set of position bits which are set in i and S_2 be S_1's complement. Now if we fix $i|j$, we can fix all the bits of j that are present in S_2, whereas all the bits of j which are in S_1 can take both 0 and 1. So, $i\&j$ can take all the possible values of submasks of i, for a fix i and $i|j$.

How to sum up the answer for a fixed i?
Let's define supermasks of i as the collection of all masks for which i is a submask.
For a fix $i|j$, which is a supermask of i, $i\&j$ can be any of the submasks of i, we'll take values such that character of S at index $i|j$ and $i\&j$ are the same. So, if c_1 and c_0 denote the count of supermasks of i which have values 1 and 0 respectively, and d_1 and d_0 count of submaks of i which have values 1 and 0 respectively, our answer for i is $c_1 \cdot d_1 + c_0 \cdot d_0$.

How to calculate these values optimally?
We can use the SOS Dynamic Programming technique to calculate the values for all i in $O(N \cdot \log N)$ time.

EXPLANATION:

If we think naively for once, we can just iterate through all possible i and j and check that $S_{i|j} = S_{i\&j}$, but this will take $O(N^2)$ time, which will exceed the Time Limit.

To optimize the approach, we can try to fix i for once and then see what happens. Now, we'll try to analyze the bits of j and how it affects the answer. Let S_1 be the set of position bits which are set in i, and S_2 be the set of positions of bits which are not set in i. If we further fix $i|j$, the bits of j that are present in S_2 get fixed, whereas all the bits of j from S_1 can take both values 0 and 1. Since in $i\&j$ is a submask of i, $i\&j$ only contains set bits from S_1, and hence $i\&j$ can be any of the submasks of i. Let's define supermasks of i as the collection of all masks for which i is a submask. For a fix $i|j$, which is a supermask of i, $i\&j$ can be any of the submasks of i, we'll take values such that S at $i|j$ and $i\&j$ are the same. If d_1 and d_0 denote count of submaks of i which have values 1 and 0 respectively, if $S_{i|j} = 1$, we have d_1 values of $i\&j$, otherwise d_0 values of $i\&j$. Let c_1 and c_0 denote the count of supermasks of i which have values 1 and 0 respectively. So, for a fixed i the total answer is $c_1 \cdot d_1 + c_0 \cdot d_0$.

We want to calculate the values of c_1, c_0, d_1, d_0 for all values of i. If we iterate through all the submasks and supermasks for every i, the time taken will be $O(3^{\log_2 n}) = O(3^{20})$. To further optimize this, we have to use the SOS Dynamic programming approach, which can calculate the sum of S at all submasks for every i in much less time.

TIME COMPLEXITY:

In the SOS Dynamic Programming Approach populating the DP values for all i will take $O(N \cdot \log N)$ time and then adding answer at each i will take $O(N)$ time. So our total time complexity will be $O(N \cdot \log N)$

PREREQUISITES:

Bitwise operations

PROBLEM:

We are given an array of N elements. From this array, we create array B of $\frac{N \cdot (N-1)}{2}$ elements where for every $1 \leq i < j \leq N$, we add $A_i \wedge A_j$ to B. Then, the following operations are performed until we are left with a single element:

- Let the current maximum and minimum element of array B be X and Y respectively.
- Remove X, Y and add $X \vee Y$ to the array B.

We need to output the final element of B.

EXPLANATION:

- The statement seems very complicated, but the important thing to notice is the bitwise operation $\vee$. The property of bitwise or is that if we are performing $X \vee Y$, bit i will be set if atleast one of bit i in X or Y is set.
- This leads us to the most crucial observation of the problem, for every bit i, if bit i is set in **atleast one** of $B_1, B_2, \ldots B_{\frac{N \cdot (N-1)}{2}}$, then bit i will be set in the final remaining element of B when all the operations are performed. We don't need to worry about how the operations are performed.
- For bit i to be set in atleast one element of B, we need to have atleast 2 elements in A say j and k where A_j and A_k both have bit i set. This sets the bit i in $A_j \wedge A_k$.
- Now we have the following solution, iterate over every valid bit i, count the number of elements in array A which have bit i set. If this count is greater than 1, the final answer will have bit i set else it will be unset.

TIME COMPLEXITY:

$O(N \log 10^9)$ for each test case.

PREREQUISITES:

Trees, Dynamic Programming

PROBLEM:

Chef is given a rooted tree with N nodes numbered 1 to N where 1 is the root node.
Each node is assigned a lowercase english alphabet between a to z (both included).

Given two nodes A and B:

- Let $LCA(A, B)$ denote the **lowest common ancestor** of nodes A and B.
- Let $DIS(A, B)$ denote the string containing all the characters associated with the nodes in the same **order** as the shortest path from node A to node B (**excluding** the character associated to node B). Note that $DIS(A, A)$ is an **empty** string.

Chef needs to answer Q queries. Each query is of the type:

- $U\ V$: Given two integers U and V $(1 \leq U, V \leq N)$, determine if there exist a concatenation of a non-empty **subsequence** of $DIS(U, LCA(U, V))$ and a non-empty subsequence of $DIS(V, LCA(U, V))$ such that it is a **palindrome**.

EXPLANATION:

The first thing that needs to be done is to pre-calculate lowest common ancestors for all nodes using binary lifting so that we can find the lowest common ancestor for any two nodes in $log(N)$ time. More details regarding that can be found **here**

Along with that, we also define another variable $cnt[node][x]$ that stores frequency of character x from $root$ till $node$. This can be again done using DFS.

Once we are done with both these preprocessing, we can solve for each query in $log(N)$ time. Say for a query we are given U and V, we can calculate their lowest common ancestor from the algorithm above. Let that be lca. Now for concatenation of a particular subsequence of $DIS(U, lca)$ and $DIS(V, lca)$ to be a palindrome they both need to have at least one character common, i.e $DIS(U, lca)$ and $DIS(V, lca)$ should have one character in common say a since then our final string can be aa which is a palindrome.
Thus for each query we loop through all characters from a to z and say for a particular character j if

$$cnt[U][j] - cnt[lca][j] > 0\ AND\ cnt[V][j] - cnt[lca][j] > 0$$

then that implies they both have character j common and so we can form a palindrome string.

TIME COMPLEXITY:

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

PROBLEM:

A string A of length N is said to be **anti-palindrome**, if for each $1 \leq i \leq N$, $A_i \neq A_{(N+1-i)}$.

You are given a string A of length N (consisting of lowercase Latin letters only). Rearrange the string to convert it into an anti-palindrome or determine that there is no rearrangement which is an anti-palindrome.

If there are multiple rearrangements of the string which are anti-palindromic, print **any** of them.

QUICK EXPLANATION

What is the condition for NO answer?

If the string is of odd length the answer must be NO, because of the centremost letter. Otherwise, intuitively we can observe that if a letter occurs more than n/2 times there can not be any rearrangement possible. Can be also proved by pigeonhole priciple.

How to rearrange if no letter appears > n/2 times in an even length string?

We can sort the string. Now we reverse the right half of the string, that way letters at a[i] and a[i +n/2] are against each other. Since no letter appears > n/2 times both these characters are different.

EXPLANATION:

For a string with odd length the answer is always NO because of the centremost character. So, from now on we'll deal with the even length strings.

If any character appears more than n/2 times, using pigeonhole principle we can conclude that no arrangement is possible. If it is not the case, we can make a arrangement by sorting the string and putting the a[i] and a[i + n/2] characters against each other. So, our rearrranged string will look like a[1], a[2], … a[n/2], a[n], a[n-1] … a[n/2 + 1].

TIME COMPLEXITY:

O(n) for each test case.

PREREQUISITES:

Basic Number theory

PROBLEM:

You are given two integers N and K.

Construct an array $A_1, A_2, \ldots, A_n$ consisting of distinct positive integers. The following conditions should be met:

- $1 \leq A_i \leq 2 \cdot 10^4$;
- there isn't any subset of A with sum K.

We can show that the answer always exists.

EXPLANATION:

Consider an integer d that does not divide K and take the first N multiples of d. It is obvious that no subset has a sum equal to K(the sum of each subset is divisible by d which does not divide K). The constraints on K allow us to find such $d \leq 19$ $(2 * 3 * 5 * 7 * 11 * 13 * 17 * 19 > 10^6)$. Thus, iterate from $d = 2$ to 19 and if d does not divide K take the first N multiples of d.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

You are given three arrays A, B, and C, all of length N. You also have two integers k_1 and k_2.

For every index $1 \leq i \leq N$, you must choose **exactly one** of A_i, B_i, or C_i. Find the **maximum** possible sum of chosen elements, such that:

- At most k_1 elements are picked from A, and
- At most k_2 elements are picked from B

EXPLANATION:

Here to solve this we will first replace $A[i]$ with $max(A[i] - C[i], 0)$ and $B[i]$ with $max(B[i] - C[i], 0)$ and add all the values in C to the current answer.
We will then take indices of k_1 largest elements in A, add all those values to the current answer and insert all the values of $(B[i] - A[i])$ (where i is an index of one of the k_1 largest values in a) into a multiset 's'. Let's denote 'rem' as an array containing all the remaining indices. We now create two sets (set of pairs) 'sa' and 'sb' where sa contains all $A[i]$ and i combined, sb contains all $B[i]$ and i combined for all i belongs to rem. Now we add k_2 elements from B greedily. Let $Smax$ be the maximum value in s and $SAmax$ be the maximum value in sa and $SBmax$ be the maximum value in sb,

- if $SAmax + Smax >= SBmax$ we remove $Smax$ from s, $SAmax$ from sa, add $(SAmax + Smax)$ to the current answer, add $B[i] - A[i]$ to s, and remove $B[i]$ from sb where i is the index of $SAmax$.
- If $SBmax > SAmax + Smax$, we add $SBmax$ to the current answer, remove $SBmax$ from sb and remove $A[i]$ from sa where i is the index of $SBmax$.

We do this operation k2 times to get our final answer.

TIME COMPLEXITY:

$O(NlogN)$ for each test case.

PROBLEM:

Given an array A consisting of N non-negative integers.

You need to perform the following operation $(N-1)$ times:

- Sort the array A in ascending order.
- Let M be the current size of the array A. Replace the array A by $[(A_2-A_1),(A_3-A_2),\ldots,(A_M-A_{M-1})]$. Note that each time the operation is performed, the size of the array reduces by one.

Find the remaining element after $(N-1)$ operations.

QUICK EXPLANATION:

We make an observation that after $O(logC)$ iterations there can be at most $O(logC)$ non-zero elements. Thus we would have an array with majority of the elements as zero. Now instead of creating large arrays with most of elements zero, we can just keep a count of zeroes and only keep the non zero elements in the array and perform operations on that array. This would drastically reduce the time to perform the operation on an array since the size of the array would be really small.

Thus we would perform operations on the array as mentioned in the question but would exclude the zero elements from the array and keep a track of it in a variable. But we would include one zero element in the array(if it exists) so as to process its immediate right neighbour.

EXPLANATION:

First we observe that after $O(logC)$ iterations there can be at most $O(logC)$ non-zero elements. Lets prove it here.

Let's say after the operation we have m non-zero numbers

$a_1 \leq a_2 \leq a_3 \leq ... \leq a_m$

then before the operation these were differences in sorted order, so the largest element is no smaller than sum of them, the second largest is no smaller than sum of them except one (the largest) and so on

so before the operation we had m non-zero numbers which are at least

$$a1 \leq a_1 + a_2 \leq a_1 + a_2 + a_3 \leq ... \leq a_1 + a_2 + ... + a_m$$

now apply this many k times

Basically we are calculating prefix sums of an array k times

Now it's easy to see that the last number before k operations should be at least $\binom{k+m-2}{k-1}$

for k = m = $log_2 C + 1$ it is bigger than C

Thus we can just take array and perform the operation as explained above in quick explanation.

TIME COMPLEXITY:

$O(NLogC)$ for each test case, where $C \leq 10^{18}$

PROBLEM

Given powers of N people standing in the queue in the array A, the following happens every second until the queue is not empty.

- The person in front of the queue uses the ATM and then leaves the queue.
- After the first step, if the queue is empty, the process stops. Otherwise, for each person from left to right (except for the first person in the queue), if the person has a **strictly greater** power level than the person in front of him, he will overtake the person in front, i.e. their positions get swapped. Note that the overtaking happens one after the other from left to right, i.e. state of the queue changes after every overtake.

For each person, find the time at which they will use the ATM.

QUICK EXPLANATION

- In i-th minute, the person with maximum power among first $min(N, 2 * i - 1)$ shall use ATM, since only first $min(N, 2 * i - 1)$ positions can reach front of queue in i seconds.
- This process can be simulated using a max heap, breaking ties by original position in the queue.

EXPLANATION

Let's assume person p has the larger power than all people in front of him. Then we can see that until p-th person is not at front of the queue, he shall always overtake in the second step. If there were x people before person p before this second, now there can be at most $x - 2$ people before him. So every second, this person moves 2 units closer to the start of the queue. This person cannot move to head any faster.

Hence, after $\lceil x/2 \rceil$-th second, this person shall be in front of the queue.

Generailization

Earlier, we tried to compute when person p reaches the front of the queue. Now, let's try finding the set of people, who can reach the front of the queue in the first i seconds.

Before the first operation, only the first person can be at the start of the queue.
Before the second operation, Any from the first three people can be at the start of the queue.
Before the third operation, Anyone among the first five people can reach start of queue.

We can generalize to see that before i-th second, anyone from the first $min(N, 2 * i - 1)$ people can reach the start of the queue if they have maximum power.

Let's consider the person with maximum power among the first $min(N, 2 * i - 1)$ people. This person always overtakes due to maximum power so this person must be at the start of the queue if not already removed before the i-th operation.

So, for i-th operation, the person having maximum power among the first $min(N, 2 * i - 1)$ people in the original queue, which is not yet removed, shall be the person at the start of the queue.

Considering example

```
5
2 3 4 1 5
```

Before the first operation, only the first person can be at the start of the queue.
Before the second operation, the person with maximum power among the first 3 people, which is the person with power 4, is at the front.

Before the third operation, the person with maximum power among the first 5 people is the person with power 5. And so on.

Implementation

Let's assume there's a data structure DS, which can handle the following operations

- Push: Insert a person with index i and power x
- Poll: Find the person with maximum power, breaking ties by index

Let's simulate operations one by one. Also, maintain P as the index of first person not included yet. Before the first operation, only the first person is added to DS. After each operation, two more people are added to DS, until all people are present in DS. For each operation, Polling the best person from DS gives the person at the start of the queue.

Above functionality is present in a data structure called **Priority Queue**, also known as Heap.

TIME COMPLEXITY

The time complexity is $O(N * log(N))$ per test case.

PREREQUISITES:

The mean of an array B of size M is defined as: $\text{mean}(B) = \frac{\sum_{i=1}^{M} B_i}{M}$.

PROBLEM:

You are given two integers N and X. Output an array A of length N such that:

- $-1000 \leq A_i \leq 1000$ for all $1 \leq i \leq N$.
- All A_i are **distinct**.
- $\text{mean}(A) = X$.

If there are multiple answers, print any. It is guaranteed that under the given constraints at least one array satisfying the given conditions exists.

As a reminder, the mean of an array B of size M is defined as: $\text{mean}(B) = \frac{\sum_{i=1}^{M} B_i}{M}$.

For example, $\text{mean}([3, 1, 4, 8]) = \frac{3+1+4+8}{4} = \frac{16}{4} = 4$.

EXPLANATION:

The problem can be divided into two cases:

Case 1: N is even. To achieve an average of X , the sum of all the values of array A must be $N \cdot X$. We simply create $N/2$ pairs such that their sum is $2 \cdot X$ each. Total sum of these pairs would be $2 \cdot X \cdot N/2 = N \cdot X$
Therefore the average of these N values is $(N \cdot X)/N = X$. One possible way to create these pairs is to pair values around X *i.e* $X - 1$ *and* $X + 1$,$X - 2$ *and* $X + 2$ and so on.
To print the answer for this case run a loop from $i = 1$ to $i = N/2$ and on each iteration print two values $X - i$ and $X + i$.
Case 1: N is odd. To achieve an average of X , the sum of all the values of array A must be $N \cdot X$. We simply create $(N - 1)/2$ pairs such that their sum is $2 \cdot X$ each and append X to the end of the array. Total sum of these pairs would be $2 \cdot X \cdot (N - 1)/2 + X = N \cdot X - X + X = N \cdot X$
Therefore the average of these N values is $(N \cdot X)/N = X$. One possible way to create these pairs is to pair values around X *i.e* $X - 1$ *and* $X + 1$,$X - 2$ *and* $X + 2$ and so on.
To print the answer for this case run a loop from $i = 1$ to $i = (N - 1)/2$ and on each iteration print two values $X - i$ and $X + i$. Then print X in the end.

The value of A_i never exceeds 600 and never drops below -500 in this approach which satisfies the given constraints.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

The average of N values is : $\frac{Sum\ of\ all\ values}{N}$

PROBLEM:

Chef has a set S containing N **distinct** integers.

Chef wants to gift Chefina an array A of any finite length such that the following conditions hold true:

- $A_i \in S\ \forall i$. In other words, each element of the array A should belong to the set S.
- **Mean** value of all the elements in A is **exactly** X.

Find whether there exists an array A of finite length satisfying the above conditions.

EXPLANATION:

Let min be the minimum of N given integers and max be the maximum of N given integers, then the average of these N integers always lies in the range $[min, max]$. Therefore if $min > X$ or $max < X$ the answer is `NO`. Otherwise the answer is always `Yes`.

Proof that the answer is always `yes` for the latter scenario:

Case 1: X is present in S, take a single occurrence of X, it has an average of X.
Case 2: X is not present in S,
Let $a = X - min$ and $b = max - X$.
Then $\frac{(a \cdot max + b \cdot min)}{(a+b)} = (\frac{(max - X) \cdot min + (X - min) \cdot max)}{(max - X + X - min)} = \frac{X \cdot (max - min)}{(max - min)} = X.$

This means we can take a occurrences of the maximum element in S and b occurrences of the minimum element in S to get an average of X.

Thus, if $min \leq X \leq max$ the answer is always `Yes` else `No`

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

It is Chef's birthday. You know that Chef's favourite number is X. You also know that Chef loves averages. Therefore you decide it's best to gift Chef 3 integers A_1, A_2, A_3, such that:

- The mean of A_1, A_2 and A_3 is X.
- $1 \leq A_1, A_2, A_3 \leq 1000$.
- A_1, A_2 and A_3 are **distinct**.

Output any suitable A_1, A_2 and A_3 which you could gift to Chef.

As a reminder, the mean of three numbers P, Q, R is defined as: $mean(P, Q, R) = \frac{P+Q+R}{3}$.

For example, $mean(2, 3, 5) = \frac{2+3+5}{3} = \frac{10}{3} = 3.33\overline{3}$, $mean(2, 2, 5) = \frac{2+2+5}{3} = \frac{9}{3} = 3$.

EXPLANATION:

There are several possible solutions to this problem. Here are two of them:

Based on Total Sum

Given that the average is X, the total sum of three elements will be $S = 3 \cdot X$.
Now, given $2 \leq X \leq 100$, we will have $6 \leq S \leq 300$.
Because we need to choose 3 **distinct** positive integers, let's choose $A_1 = 1, A_2 = 2$. Now, $A_3 = S - 3$, so $3 \leq A_3 \leq 297$, which satisfies all the constraints.

Based on Average

We want to make the average of three elements to X. One way to achieve this is to symmetrically distribute elements around X.
So, we can have one of the elements as X itself. Now, we have two remaining elements which we want to assign symmetrically, so we can have one of the element as $X - 1$ and other as $X + 1$.
This assignment satisfies all the constraints.

TIME COMPLEXITY:

$O(1)$ for each test case.

PREREQUISITES:

0/1 BFS, Greedy, Dynamic Programming, Sliding Window

PROBLEM:

Given a grid of size $N \times M$. K cells of the grid are black, and the rest are white. The cost of a path from $(1, 1)$ to (x, y) (where we move one cell down/right in each step) is equal to the number of times we move between cells of different colors.

Answer Q independent queries of the form (x, y) - minimum possible cost of a path from $(1, 1) \rightarrow (x, y)$, after inverting the colors of all cells in some set of columns.

EXPLANATION:

Observation: In any optimal path from $(1, 1) \rightarrow (x, y)$, we always move between similar colored cells **when taking a step to the right**. That is, we can invert columns in such a way that we incur no cost when moving to the right, and this never affects the cost of moving downwards.

With this key observation, the problem is reduced to calculating the minimum number of times we must move **downwards** between different colored cells in the original grid to reach the destination cell.

Hint 1

Given below is a naive DP approach to the problem.

Let $dp[i][j]$ represent the least cost over all paths from $(1, 1) \rightarrow (i, j)$. Then,

$$dp[i][j] = \min(dp[i][j-1], dp[i-1][j] + c)$$

where $c = 1$ if cells (i, j) and $(i-1, j)$ are differently colored, and 0 otherwise.

This solution however, TLE's for the given constraints.
But we can optimise! Observe that $c = 1$ in only $\approx 2 \cdot K$ states of the above DP, and computation of the other states are redundant. Use this information and rewrite the solution.

Hint 2

Call a cell (i, j) *special* if the cells (i, j) and $(i+1, j)$ have different colors.
Also, let $s(i, j)$ equal the smallest $x \geq i$ such that cell (x, j) is special.

Observation: $dp[i][j] = dp[i+1][j] = \cdots = dp[s(i, j)][j]$.

Why?

By the definition of s, it is clear that all cells in the range $(i, j) \rightarrow (s(i, j), j)$ are of the same color. Moving down from (i, j) to any of the aforementioned cells doesn't increase the cost, and hence $dp[i+1][j], \ldots dp[s(i, j)][j] \leq dp[i][j]$.

To then show that $dp[i+1][j], \ldots, dp[s(i, j)][j] = dp[i][j]$, we make use of the following observation.

Observation: $dp[i][j] \leq dp[x][j]$ for all $x \geq i$.

How?

Let $P = DRRDDRD \ldots$ be the sequence of steps (down/right) taken in the optimal path from $(1, 1) \rightarrow (x, j)$. Erase the last $x - i$ down steps (D's) from P. Then, invert the columns such that all R steps have cost 0.

Now, this new sequence of steps reaches cell (i,j). The cost of every R step is 0. Since inverting the columns doesn't change the cost of D steps, and the total number of times we changed colors in D steps equals $dp[x][j]$, the new sequence also changes colors atmost $dp[x][j]$ times.

Thus, we have found a path from $(1,1) \rightarrow (i,j)$ with cost $\leq dp[x][i]$ which implies $dp[i][j] \leq dp[x][j]$.

Hint 3

From the observations made in the previous spoiler, we know that the answer to any query (x,y) is equal to $dp[s(x,y)][y]$ (or $dp[N][y]$ if $s(x,y)$ doesn't exist). Thus, all we have to do is compute $dp[i][j]$ **for only** *special* cells (i,j).

How will you accomplish this?

Hint

Try constructing a graph with weighted edges between *special* cells.

Solution

Firstly, generate the list P of all special cells. This can be done trivially in $O(K)$. Also, for convenience, add cells $(N,1),(N,2),\ldots,(N,M)$ to this list.

Create a graph of $|P|$ nodes, each corresponding to a particular special cell.

The cost of moving from special cell (i,j) to cell $(s(i,j),j)$ is 1; add a directed edge between these nodes with weight 1. Also, since we may move to the left with cost 0, add a directed edge from (i,j) to each of the nodes $(s(i,j+1),j+1),(s(i,j+2),j+2),\ldots$ with weight 0.

Running **Dijkstra**/**0/1 BFS** on the generated graph (after appropriately adding a node corresponding to cell $(1,1)$) will give us the minimum possible cost of a path to each of the special cells, which can be used to answer the queries.

But wait! The number of edges in the above generated graph is $\approx |P|^2$, which is too much for the given constraints.
To remedy this, for each (i,j), we insert only one edge $(s(i,j+1),j+1)$ with weight 0. Then, after running dijkstra/bfs on the graph - let the mininum cost to reach node (i,j) be $f(i,j)$ - do:

$$dp[i][j] = \min(f(i,j),f(s(i,j),j),f(s(s(i,j)),j),\ldots)$$

Why does this work?

We reason with the help of a simplified example.

Consider 3 special cells $(a,1),(b,2),(c,3)$ where $a \leq b,c$.
Now, our above approach adds an edge from $(a,1)$ to only $(b,2)$. We go on to show that $(b,2)$ adds an edge that *implicitly* propogates a 0 edge from $(a,1) \rightarrow (c,3)$.

Case 1**:** $b \leq c$
Here, node $(b,2)$ adds a 0 weight edge to $(s(b,3),3) = (c,3)$. Thus, running dijkstra on this graph would give us $f(c,3) = 0$, which is what we'd have got if we added an edge from $(a,1) \rightarrow (c,3)$.

Case 2**:** $b > c$.
Here, cell $(b,2)$ adds a 0 weight edge to $(s(b,3),3) = (N,3)$. Then, running dijkstra would give us $f(N,3) = 0$.
By our 2^{nd} observation in "Hint 2", we know that $dp[c][3] \leq dp[N][3]$. Thus $dp[c][3] = \min(f(c,3),f(N,3)) = 0$, which is the answer we were looking for.

A more robust proof of the approach is left to the reader as an exercise 😛.

Refer my attached code for implementation details.

TIME COMPLEXITY:

There are atmost $P = M + 2 \cdot K$ *special* cells. Computing $s(i,j)$ can be done in $O(\log P)$ using binary search.
Since there are $2 \cdot P$ nodes in the generated graph, running BFS takes $O(2 \cdot P)$ time.
Updating $dp[i][j]$ as the minimum of the computed distance of special cells below cell (i,j) can be done in $O(P)$ using sliding window.
Finally, answering each of the queries takes $O(\log P)$ per query.

The total time complexity is therefore:

$$O(P \log P + 2 \cdot P + P + Q \log P) \approx O((P + Q) \log P)$$

per test case.

PREREQUISITES:

Inversions

PROBLEM:

Chef is given a binary string A of length N. He can perform the following operation on A any number of times:

- Choose L and R $(1 \leq L \leq R \leq N)$, such that, in the substring $A[L, R]$, the number of \$1\$s is **equal** to the number of \$0\$s and **reverse** the substring $A[L, R]$.

Find the lexicographically **smallest** string that Chef can obtain after performing the above operation **any** (possibly zero) number of times on A.

String X is lexicographically smaller than string Y, if either of the following satisfies:

- X is a prefix of Y and $X \neq Y$.
- There exists an index i such that $X_i < Y_i$ and $X_j = Y_j, \forall j$ such that $1 \leq j < i$.

EXPLANATION:

The first intuition when we read the problem is, can we sort the string? If Yes, we have got our answer, as the sorted string is the lexicographically smallest possible string that we can get.

Let's see if we can sort the string. Let there be an index i such that $S_i = 1$ and $S_{i+1} = 0$. If there is no such index i, then we have already sorted our string. Otherwise, the substring S_iS_{i+1} is a valid substring, and we can reverse it, making it 01. We can repeat this process as long as we can find such index i, and hence finally ending up with a sorted string.

But how to prove that the above process will eventually terminate?

TIME COMPLEXITY:

We need to sort the given string - $O(N \cdot \log N)$ for each test case.

PREREQUISITES:

Trees, Dynamic programming, Depth First Search

PROBLEM:

We are given a tree with N nodes where each node is black or white in color. In one step, we can choose a vertex u and toggle u along with all the neighbors of u.

We are asked to find the minimum number of steps required to make all the nodes black or report that it is not possible.

QUICK EXPLANATION:

- We can initially root the tree at some vertex.
- Let color 0 be black and color 1 be white.
- Let us define an $N \cdot 2 \cdot 2$ dp. Let Let $dp[u][state][toggle]$ denote the minimum number of steps required to make the color of vertex u equal to $state$ and all the other vertices of subtree of u equal to black and also $toggle$ denotes whether one step is applied on vertex u or not.
- This dp can be calculated by depth first search with bottom-up approach.
- If vertex u is a leaf, then $dp[u][color[u]][0] = 0$ and $dp[u][color[u] \oplus 1][1] = 1$.
- While calculating $dp[u][state][toggle]$, first we need to figure out what is the final state of every direct child of u must be depending on $toggle$. Let it be fin.
- After that, for every child x we need to take the minimum values of child dp states $dp[x][fin][0]$ and $dp[x][fin][1]$. Let the parity of total child toggles taken here be par.
- While calculating $dp[u][state][toggle]$, we also need to figure out what is the parity of the total number of direct child toggles needed in order to make the color of u equal to $state$. If this value is equal to par, we are done. Else for exactly one child x, we need to take the maximum of $dp[x][fin][0]$ and $dp[x][fin][1]$ instead of minimum and update the dp state accordingly.

EXPLANATION:

Firstly, make the tree rooted by fixing some vertex as root.

The first observation is that we do not apply more than one step at some vertex. This is because for suppose if we apply two steps, it is the same as not applying any step at all.

Let color 0 represent black and color 1 represent white.

Also, let $color[u]$ denote the color of vertex u.

This problem can be solved by dynamic programming. For this, let us define a state. Let $dp[u][state][toggle]$ **denote the minimum number of steps required to make the color of vertex u equal to $state$ and all the other vertices of subtree of u equal to black and also $toggle$ denotes whether one step is applied on vertex u or not.**

It is important to note that, while calculating the dp state of u, we only consider the subtree of u and ignore everything else.

Let us intialize all the values in the dp state to infinity.

Let us first see the base case.

- Let u be a leaf. Now it takes 0 operations to make color of u equal to $color[u]$ and 1 operation with toggle to make color of u equal to $color[u] \oplus 1$. (Here xor is used just for showing the other color by toggling the current color).
- Therefore, the dp transitions for vertex u being a leaf are $dp[u][color[u]][0] = 0$ and $dp[u][color[u] \oplus 1][1] = 1$.

Now let us try to see how to make the transitions for vertex u being not a child. Let us assume $state = 0$. (The similar case work can be done for $state = 1$). Let us also assume that the initial $color[u] = 0$. (The similar case work can be done for $color[u] = 1$).

Therefore, we are now trying to calculate $dp[u][0][0]$ and $dp[u][0][1]$ where $color[u] = 0$.

Case 1: $toggle = 0$

- This means that we haven't applied any step on vertex u.
- Also, according to our dp assumption every vertex in the subtree of u other than it must be black i.e, color 0.
- Therefore iterate over every child x of u, and take the **minimum** of $dp[x][0][0]$ and $dp[x][0][1]$ and add it to $dp[u][0][0]$.
- But we are not done yet. Since $color[u] = 0$ initially, we need to apply a total of **even number of steps** on the direct children of u which share an edge with it in order to keep $color[u] = 0 = state$.
- By taking the minimums as explained above, if the number of children toggles are even, then we are done. If it is odd, we need to pick **exactly one child** x of u and do the following: We need to change our previous decision and take the **maximum** of $dp[x][0][0]$ and $dp[x][0][1]$ instead of minimum and update $dp[u][0][0]$ accordingly. And we need to take such a child x which incurs the minimum extra cost to $dp[u][0][0]$.

Case 2: $toggle = 1$

- This means that we have applied exactly 1 step on vertex u.
- Also, according to our dp assumption every vertex in the subtree of u other than it must be black i.e, color 0.
- Therefore iterate over every child x of u, and take the **minimum** of $dp[x][1][0]$ and $dp[x][1][1]$ and add it to $dp[u][0][1]$. Here, unlike the previous case, we want the state of x to be equal to 1 because anyways later it will become equal to state 0 by the step/toggle applied on vertex u.
- Since $color[u] = 0$ initially, we need to apply a total of **odd number of steps** on the direct children of u which share an edge with it in order to make $color[u] = 1$ and after a toggle on u it becomes $color[u] = 0 = state$.
- By taking the minimums as explained above, if the number of children toggles are odd, then we are done. If it is even, we need to pick **exactly one child** x of u and do the following: We need to change our previous decision and take the **maximum** of $dp[x][1][0]$ and $dp[x][1][1]$ instead of minimum and update $dp[u][0][1]$ accordingly. And we need to take such a child x which incurs the minimum extra cost to $dp[u][0][1]$.

- Finally, we need to add 1 to $dp[u][0][1]$ for the toggle we are applying on vertex u.

Calculating these dp states can be done in a bottom up manner by using depth first search from the root.

You can have a look at the code for better understanding.

TIME COMPLEXITY:

$O(N)$ for each testcase.

PROBLEM

Given N and S, consider an array of length N where $A_i = i$, find position $x(1 \leq x \leq N)$ such that $\sum_{i=1}^{x-1} A_i + \sum_{i=x+1}^{N} A_i = S.$

If there are multiple such x, print any one of them. If no such x exists, print -1.

QUICK EXPLANATION

- For chosen x, $\sum_{i=1}^{x-1} A_i + \sum_{i=x+1}^{N} A_i$ can be written as $\sum_{i=1}^{N} A_i - x$. Since $A_i = i$, $\sum_{i=1}^{N} A_i = \frac{N * (N+1)}{2}$
- So, we can only choose $x = \frac{N * (N+1)}{2} - S$.
- If $1 \leq x \leq N$ is satisfied, chosen x is the answer, otherwise no such x exists.

EXPLANATION

Let's focus on the expression first.

$$\sum_{i=1}^{x-1} A_i + \sum_{i=x+1}^{N} A_i = S.$$

If we consider both summations, we can see that only element not added is A_x. All elements before x-th element is included in the first summation, while all elements after x-th element are covered in second summation.

So we can write $\sum_{i=1}^{x-1} A_i + \sum_{i=x+1}^{N} A_i = \sum_{i=1}^{N} A_i - A_x.$

We can also replace A_i by i, so we have $\sum_{i=1}^{N} i - x = S.$

The first summation is just sum of first N natural numbers, which is $\frac{N * (N+1)}{2}$. Hence the expression becomes $x = \frac{N * (N+1)}{2} - S$. This way, we have computed the required x.

We also require $1 \leq x \leq N$, since the element with value x has to exist in the array. Hence, if the condition is not satisfied, there's no such x which can satisfy the requirements. Otherwise found x is the required answer.

TIME COMPLEXITY

The time complexity is $O(1)$ per test case.

PROBLEM

Given N binary strings, each of length M. Concatenate all N strings in some order in a single string T of length $N * M$, aiming to minimize the number of inversions in T.

QUICK EXPLANATION

- The optimal order would be sorting the strings in nondecreasing order of number of ones present in S
- We can first build the string and then calculate the number of inversions on the concatenated string.

EXPLANATION

Solving for $N = 2$

Let's say we have two strings A and B, which we need to concatenate while minimizing the number of inversions. We can try both AB and BA and pick the one with fewer inversions.

Let's denote $C_{S,c}$ denote the number of occurrences of character c in S and the number of inversions in S by $f(S)$.

We can denote $f(A + B) = f(A) + f(B) + C_{A,1} * C_{B,0}$ and $f(B + A) = f(B) + f(A) + C_{B,1} * C_{A,0}$

Types of inversions

Let's call inversion pair (x, y) if 1 appears at position x and 0 appears at position y and $x < y$.

We can divide inversions into two categories

- **Inversions within the same string**
  This includes inversions where both x and y lie on the same string. Irrespective of where this string is concatenated, this pair shall always exist as an inversion. We cannot change the number of inversions within a string.
- **Inversions across strings**
  When solving for $N = 2$, when we concatenated AB, there must be some pairs where 1 appeared in A and 0 appeared in B, forming an inversion in concatenated string. These inversions are dependent on the order of strings we choose.

Our aim is to reduce the inversions of the second type since the first type of inversion never changes with the order of strings.

Deciding the order of a pair of strings.

Let's say we have already chosen the order of strings to be concatenated, and we are only allowed to swap adjacent strings. For some i, we need to decide whether swapping S_i and S_{i+1} is beneficial or not.

Observation: Only the number of inversion between strings S_i and S_{i+1} is affected by this swap.
Proof: Considering string S_j for $j < i$, both S_i and S_{i+1} appear to the right of string S_j, so no 0 or 1 from right to left of S_j, leaving the number of inversions arising from string S_j unaffected.

Similarly, if we have string S_j where $j > i + 1$, then also, S_{j+1} is to the right of both S_i and S_{i+1}. So the number of inversions arising from S_j are also unaffected by the swap.

Hence, we can decide which one of S_i or S_{i+1} should come first sorely based on the number of inversions in string $S_i + S_{i+1}$ and string $S_{i+1} + S_i$.

Observation: In the optimal order of strings, there doesn't exist any beneficial swap, as beneficial swap reduces the inversions, but our string is already optimal.

Choosing the order of strings

Now, for an adjacent pair, we know whether the swap would be beneficial. So we can actually simulate bubble sort since bubble sort swaps elements until the array is sorted. We can sort the strings by defining a comparator function, accepting two strings A and B, and comparing $f(A) + f(B) + C_{A,1} * C_{B,0}$ with $f(A) + f(B) + C_{B,1} * C_{A,0}$ to decide which string should appear before which.

Since bubble sort is slow, we can use sort algorithms like merge sort to sort them efficiently.

Computing the number of inversions

Now that we have the computed binary string, we need to count inversions. Counting inversions in an array is a well-known problem, but for binary strings, it can be solved even faster.

Let's say we iterate on string S from left to right. We have two variables, ans, and cnt_1, denoting the number of inversions found yet, and the number of 1.

- If the current character is 0, this position shall form an inversion pair with all occurrences of 1 before the current position. It can be written as $ans = ans + cnt_1$
- If the current character is 1, the number of 1s should be incremented. which implies cnt_1 increases.

TIME COMPLEXITY

The time complexity is $O(M * N * log(N))$ per test case due to sorting.

PREREQUISITES:

Greatest Common Divisor, Floor function

PROBLEM:

Given an array A of N integers, we can apply the following operation on A any number of times (possibly zero): Choose an index i where $A_i \geq 1$ and perform $A_i = \lfloor \frac{A_i}{2} \rfloor$.

We need to find the minimum number of operations required to make $GCD(A_1, A_2 \ldots A_N)$ odd.

QUICK EXPLANATION:

- Let us first find the gcd of all the elements and store it in gcd.
- In order to make gcd odd, we need to find the maximum value x for which gcd is divisible by 2^x and apply the operation x number of times on that element in the array which is divisble by 2^x but not 2^{x+1}.
- Thus, x will be our final answer.

EXPLANATION:

- For each element A_i in the array, let us calculate pow_i which is defined as follows: pow_i is the value such that A_i is divisible by 2^{pow_i} but not 2^{pow_i+1}. In other words, pow_i is the maximum power of 2 which divides A_i.
- In order to make the gcd of all elements odd, it is enough that we make one of the elements odd. For that, we need to remove 2^{pow_i} from some element A_i by performing pow_i operations on it.
- To minimize the number of operations, we need to select such an index i for which pow_i is minimum.
- Thus, our final answer will be $min(pow_1, pow_2, pow_3, \ldots pow_N)$.

TIME COMPLEXITY:

$O(N \log 10^9)$ for each testcase.

PREREQUISITES:

Count of a unique element in any range of an array using concept similar to prefix sum .

PROBLEM:

You have a binary string S of length N.

You must perform the following operation on the binary string S **exactly once**:

- Choose two integers L and R $(1 \leq L \leq R \leq N)$ and invert the **substring** $S_{L \ldots R}$ (i.e change 1 to 0 and change 0 to 1).

Determine whether you can make the number of zeroes in S equal to number of ones in S by performing the above operation exactly once. If there exists a way, also output the bounds of the chosen substring.

QUICK EXPLANATION:

Whenever the length of the string is even it is possible to make the count of $zeroes$ and count of $ones$ equal by inverting some prefix of the string.

EXPLANATION:

If N is odd the count of $zeroes$ and $ones$ in the string can never be made equal as addition of two same numbers is even. Therefore when N is odd the answer is always `NO`. Now, we will prove that when N is even it is always possible to invert some prefix of the string to make the count of $ones$ and the count of $zeroes$ in the string equal.
Let P_i represent the prefix of the string S ending at index i $i.e.$ substring formed by the first i characters of the string $S \rightarrow S[1.....i]$. Let the difference between the count of $ones$ and the count of $zeroes$ in string S be a. Suppose we invert the complete string S the difference now becomes $-a$. Also, Inverting P_1 changes a by 2 and Inverting P_2 changes the difference obtained by inverting P_1 again by 2. Overall, Inverting P_{i+1} changes the difference obtained on inverting P_i by 2. This means by inverting prefixes of different length we can make difference equal to all values from $-a$ to a where adjacent values differ by 2. Let $a > 0$ then some possible differences are $-a, -a+2.....a-2, a$.
Also as the length of the string S is even therefore either both the count of $zeroes$ and count of $ones$ in string S are even or they both are odd. This means that initial difference a is an even number. This means that the possible difference take the value of all even numbers between $-a$ to a which includes 0.
Lets $Ones_i$ represent the numbers of 1 in the prefix of length i. We just need to find an index i such that inverting P_i makes the count of $zeroes$ and the count of $ones$ equal to $N/2$. This boils down to finding an index i such that $Ones_n - N/2 = 2 \cdot Ones_i - i$.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

You are given a binary string S of length N.

You have to perform the following operation **exactly** once:

- Select an index i $(1 \leq i \leq N)$ and delete S_i from S. The remaining parts of S are concatenated in the same order.

How many **distinct** binary strings of length $N-1$ can you get after applying this operation?

QUICK EXPLANATION:

- Let str_i represents the string obtained after deleting the i^{th} character from the string. Compare str_i and str_{i+1}. When are they same? When are they different? What if we consider str_i and str_j?

EXPLANATION:

Let us use the notations as defined in the Quick Explanation and try to answer the questions that are asked there.

Consider a continuous block of $0s$. If we remove any 0 from this continuous block, then the resulting strings will be equal. The same holds for a continuous block of $1's$. So, if $S_i = S_{i+1}$, then the resulting str_i will be equal to str_{i+1}.

This motivates us to consider the string as the blocks of $0s$ and $1s$. Let us represent the string as $S = O_1Z_1O_2Z_2...O_KZ_K$, where Z_i represents a block of $0s$ and O_i represents a block of $1s$. (It is possible that string starts with 0 or ends with 1, all these cases are equivalent, so the above representation is good enough for us).

We have seen that if we remove any 1 from O_i, the resulting strings will be equal. The only case to consider is, if we remove 1 from O_i in one case and from O_j in second case $(i \neq j)$.
The first string will be: $O_1Z_1...Z_{i-1}(O_i-1)Z_i...O_jZ_j...O_KZ_K$
The second string will be: $O_1Z_1...Z_{i-1}O_iZ_i...(O_j-1)Z_j...O_KZ_K$

We can see that strings differ at the O_i block. So these both strings are different.
Therefore, total number of **distinct** strings that we can obtain is equal to the number of continuous blocks of $0s$ and $1s$.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

The BSRand machine takes a binary string A of length N as input and produces another string B of length N as follows:
For each i from 1 to N,

- Choose a random index $1 \leq j \leq N$
- Choose a random index $1 \leq k \leq N$
- Set $B_i := A_j \bigoplus A_k$

You are given two binary strings A and B. The BSRand is applied T times successively on A. What is the probability that the resulting string is B?

QUICK EXPLANATION

- Build an $(N+1) \times (N+1)$ matrix M such that for $0 \leq i,j \leq N$,

$$M_{i,j} = \binom{N}{j} \cdot p_i^j \cdot (1-p_i)^{N-j}$$

where $p_i = \frac{2i(N-i)}{N^2}$.

- Suppose A contains x ones and B contains y ones. The final answer is then $(M^T)_{x,y}$ divided by $\binom{N}{y}$.

EXPLANATION:

Let's analyze what happens when one iteration of the BSRand is run.
Each character of the output string can be treated independently, so let's focus just on the first one.
What is the probability that this character is a `'1'`?
For this to happen, there are two possibilities:

- $A_j = 1$ and $A_k = 0$, or
- $A_j = 0$ and $A_k = 1$

If A has x ones, because j and k are chosen uniformly randomly, the probability of the first case happening is $x \cdot (N-x)/N^2$. The second case similarly has the exact same probability.

So, the probability that the first character is a `'1'` is $p = 2x \cdot (N-x)/N^2$.

This applies to every single character of the output string. Thus, given a string B with y ones, the probability that the output string is B is $p^y(1-p)^{N-y}$.

So, if $T = 1$ we have solved the problem. However, this process can't be repeated for $T > 1$, since BSRand is applied once, we no longer know the exact number of ones in the string.

But we don't need to! Note that N is small, so we use that to our advantage.

ENTER MATRICES

Note that above, when we calculated the probability that the output string had y ones, we didn't really care about what exactly the input string was — we only used the fact that it had exactly x ones.

Also note that we computed the probability that a specific string with y ones was formed. We can relax this condition a little and compute just the probability that the output string has y ones. There are $\binom{N}{y}$ such strings,

each being equally likely.

Under this relaxed condition, suppose we are able to calculate the probability that the final string has y ones. Then we can simply divide this by $\binom{N}{y}$ to get the final answer, so we'll concentrate on how to do the first part.

Notice that we have already, in some sense, completely described the working of the BSRand. If it is given an input string with x ones, the probability that the output string has y ones is $\binom{N}{y} \cdot p^y(1-p)^{N-y}$, where $p = 2x \cdot (N-x)/N^2$.

Suppose we create an $(N+1) \times (N+1)$ matrix M, where $M_{x,y}$ is the value described above for x and y.
Suppose we also have a $(N+1) \times 1$ vector v, where v_i is the probability that the input string to BSRand has exactly i ones.
Let $w = Mv$. Notice that w is then a $(N+1) \times 1$ vector, where w_i is the probability that the output string has exactly i ones.

But then we can simply use w as the input probability vector, which means that $Mw = M^2v$ gives us the probability distribution of output strings when BSRand is applied twice.

Generalizing this, we can see that applying BSRand T times simply corresponds to $M^T v$.

In our case, the probability vector is such that $v_i = 0$ when $i \neq x$, and $v_x = 1$ (where x is the number of ones in A).
It's easy to see that multiplying M^T by this vector and taking the y-th entry of the result is the same as $(M^T)_{x,y}$. (where y is the number of ones in B).

M^T can be computed in $\mathrm{O}(N^3 \log T)$ by using binary exponentiation, and each entry of M can be computed in $\mathrm{O}(1)$ or $\mathrm{O}(\log MOD)$ depending on how **modular inverses** are dealt with; either way the exponentiation makes the complexity $\mathrm{O}(N^3 \log T)$.

Finally, don't forget to divide $(M^T)_{x,y}$ by $\binom{N}{y}$ to account for the fact that we want a specific string.

TIME COMPLEXITY:

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

PREREQUISITES:

String, Math

PROBLEM:

Given a binary string S, in one operation we can select any substring of size M and flip all its characters. There are Q queries and we can append a 0 or 1 character at the beginning of S in each query, what is the minimum number of operations required to make the string *lexicographically minimum* after each query.

EXPLANATION:

Below is a simple algorithm to find the minimum number of moves for each query by making the string *lexicographically minimum* :
– Iterate over string as: for i from 1 to $|S| - m + 1$ (from MSB to LSB)
– If $S[i] = 1$ then flip all $S[j]$ for j in $[i, i + m - 1]$ (all m bits) and add 1 to the ans.

This solution will be too slow (Time Complexity: $O(N * Q)$), but the idea will be helpful to reach the actual solution. Before we move to actual solution let's define few terms:
– s_i = i^{th} character of string S
– $g_i = 1$ if the i^{th} bit of the string will be flipped
– $x_i = \bigoplus(s_t)$ such that $i\%m = t\%m$ and $t \leq i$.

We will append Q zeroes to the beginning of S initially and will update them with query, we can see from the above algorithm that appending $0's$ to the beginning of string doesn't affect the answer.

Now,
$g_i = s_i \oplus (g_{i-1} \oplus g_{i-2} \oplus ... \oplus g_{i-m+1}), \quad i \leq n + 1 - m$
$\implies g_{i-1} = s_{i-1} \oplus (g_{i-2} \oplus g_{i-2} \oplus ... \oplus g_{i-m})$

$\therefore \quad g_i \oplus g_{i-1} = s_i \oplus s_{i-1} \oplus g_{i-m} \oplus g_{i-1}$
$\implies g_i = s_i \oplus s_{i-1} \oplus g_{i-m}$

Similarly expanding the term g_{i-m},
$g_{i-m} = s_{i-m} \oplus s_{i-m-1} \oplus g_{i-2m}$

$\therefore \quad g_i = (s_i \oplus s_{i-m} \oplus ... \oplus s_{i-rm}) \oplus (s_{i-1} \oplus s_{i-m-1} \oplus ... \oplus s_{i-rm-1}) \oplus g_{i-(r+1)m}$

If we expand this to the last term we get:
$g_i = x_i \oplus x_{(i-1)}$

Now the answer to the each query will be sum of all g_i's.

As stated earlier we had appended Q zeroes to the beginning of S initially and will update them with query.

Handling update part can be done as follows:
Let the index of string where update happens be cur_idx.
If 0 is appended to the front of string then no change is observed in any of the g_i.
If 1 is appended to the string the g_i value of all positions i, such that:

- $i \geq cur\_idx$ and
- $i\%m = cur\_idx\%m$ or $(i + 1)\%m = (cur\_idx)\%m$
  is flipped.

As we need sum of all g_i and as cur_idx decreases with queries, we can store the required information to handle queries in the following arrays:

- $total[0..m-1]$: $total[r]$ stores number of index i such that $i\%m = r$ and $i \geq cur\_idx$
- $plus[0...m-1]$: $plus[r]$ stores number of index i such that $i\%m = r$ and $i \geq cur\_idx$ and $g_i = 1$.

PREREQUISITES:

Familiarity with comparing string lexicographically will be useful.

PROBLEM:

You are given a string S of length N, containing lowercase Latin letters. You are also given an integer K.

You would like to create a new string S' by following the following process:

- First, partition S into exactly K non-empty **subsequences** $S_1, S_2, \ldots, S_K$. Note that every character of S must be present in exactly one of the S_i.
- Then, create S' as the concatenation of these subsequences, i.e, $S' = S_1 + S_2 + \ldots + S_K$, where $+$ denotes string concatenation.

Determine the **lexicographically** smallest S' that can be created.

EXPLANATION:

We can divide the solution in two parts, when $K = 2$ and when $K > 2$.

Creating lexicographically smallest string when $K > 2$

Let α represents the lexicographically smallest character, and β represents the second lexicographically smallest character in the string.

We want to choose a subsequence such that the resulting string is lexicographically smallest. To do this, we will greedily choose all the occurrences of α in our first subsequence. Let the last occurrence of this character be ind.

What to choose from the remaining string after choosing the all the lexicographically smallest characters?

We will start choosing all the occurrences of β that occurs after the index ind. If β also occurs before the index ind, then we are done with this subsequence. Otherwise, we have selected all the occurrences of β, and continue with the third lexicographically smallest character.

Once we have chosen all the characters for the first subsequence, we can remove these characters from the string, decrease K by 1, and continue our problem.

Creating lexicographically smallest string when $K = 2$

We want to choose a subsequence such that the resulting string is lexicographically smallest. To do this, we will first consider the lexicographically smallest character in the string. We will greedily choose all these characters in our first subsequence.

What to choose from the remaining string after choosing the all the lexicographically smallest characters?

Suppose till now, we have chosen the complete prefix of our string as part of first sub-sequence. In that case, we have a remaining smaller string and we can solve it by starting the logic from the beginning.

Otherwise, we have atleast one character that we have not taken in the sub-sequence. The first such character will be at the beginning of the second subsequence. Let us represent this character by α. Now, we consider one of the character β of the remaining string.
If that character is lexicographically greater than α, we will not take that character in the first subsequence greedily, so that we are not placing it before α in the final string.
Now, consider the case when β is lexicographically smaller than α. If β is the lexicographically smallest element in the remaining string, we will take β in the first subsequence. Otherwise, we will skip this character in order to take the lexicographically smaller character in the sub-sequence.

Note that the condition subsequences should be non-empty is redundant. If we can find a solution when empty sub-sequences are also allowed, we can convert that solution into non-empty sub-sequences by breaking down our previous sub-sequences.

TIME COMPLEXITY:

We need at most 26 non-empty subsequences to get our answer. So we can implement the above approach in $O(N \cdot 26)$ for each test case.

PREREQUISITES:

Bitwise-xor

PROBLEM:

Given a number N, we need to find the number of distinct valuyes possible for $i \oplus j$ where $1 \leq i,j \leq N$.

EXPLANATION:

- If $N = 1$, the possible xor values are $1 \oplus 1 = 0$ which has 1 unique value.
- If $N = 2$, the possible xor values are $1 \oplus 1 = 0, 1 \oplus 2 = 1, 2 \oplus 2 = 0$ which has 2 unique values.
- For the remaining cases, we consider $N \geq 3$.
- Let us define x as the highest power for which $2^x \leq N$.
- Suppose $2^x < N$. I claim that we can get all the numbers from 0 to $2^{x+1} - 1$. This can be achieved by the following way:
- For the numbers num which have bit x set and are greater than 2^x, it can be formed by $(2^x, 2^x \oplus num)$. For example, let $N = 12$, then we have $x = 3$. Number 12 (1100 in binary) can be formed by $(2^3(1000), 4(0100))$.
- Number 0 can be formed by $(1, 1)$ since $1 \oplus 1 = 0$. Number 1 can be formed by $(2, 3)$ since $2 \oplus 3 = 1$. For the remaining numbers $1 < num \leq 2^x$, we can simply get them as $(1, num \oplus 1)$. For example, $2 = 1 \oplus 3, 3 = 1 \oplus 2$ and so on.
- By the property of xor, $num \oplus 1$ is either $num + 1$ or $num - 1$, so if $1 < num \leq 2^x < N$, $1 \leq num \oplus 1 \leq N$. Hence these pairs of numbers will always be valid.
- Now what happens if $2^x = N$? All of the above cases hold true except for the case of $num = 2^x$. We cannot get this from any xor pair (i,j) where $1 \leq i,j \leq 2^x$. Since the only number with bit x set is 2^x, we must keep $i = 2^x$. Then for $i \oplus j = 2^x$, we must keep $j = 0$, which we cannot do since $j \geq 1$. Hence, in this case, except 2^x, we can get xor pair for any number from 0 to $2^{x+1} - 1$.

TIME COMPLEXITY:

$O(\log N)$ for each testcase.

PROBLEM:

Given a binary string of S of length N. For every occurence of 01 in S, a tax of X rupees will be charged, while for every occurence of 10 in S, a tax of Y rupees will be charged.

Example

For $X = 3, Y = 5$ and $S = 100101$, then S has 2 occurrences of 10 and 2 occurrences of 01, so the tax charged $= 2 \cdot 3 + 2 \cdot 5 = 16$

Rearrange the string S in any way you want. **Minimize** the amount of tax needs to be paid.

EXPLANATION:

Given that for every occurrence of 01 in S, a tax of X rupees is charged, while for every occurrence of 10 in S, a tax of Y rupees is charged. We can rearrange the string in any way we want.

Observation 1
If the given string contains only $0's$ or only $1's$ then there will be 0 occurrences of 10 and 01 in both the cases. Therefore in this case tax charged will always be 0.

Example

- $X = 2, Y = 3, S = 00000$

Tax charged $= 0$.

- $X = 2, Y = 3, S = 11111$

Tax charged $= 0$.

Observation 2
In order to **minimize** the amount of tax needs to be paid the number of occurrences of 10 and 01 in the string should be **minimized**. In order to do that either place all the $1's$ at the start of the string or at the end of the string.

CASE 1 : Placing all the $1's$ at the start of the string ensures that atmost 1 occurrence of 10 and no occurrence of 01 will be present in the string for which the tax charged will be Y rupees.

Example

- $X = 2, Y = 3, S = 1010110$

Initially there are 3 occurrences of 10 and 2 occurrences of 01; therefore, the tax charged will be $2 \cdot 2 + 3 \cdot 3 = 13$.

If we place all the $1's$ at the start of the string, the string S will become 1111000. Now the number of occurrences of 10 will become 1 and number of occurrences of 01 will become 0. Therefore, the tax charged will be $0 \cdot 2 + 1 \cdot 3 = 3$.

CASE 2 : Similarly, placing all the $1's$ at the end of the string ensures that atmost 1 occurence of 01 will be present in the string for which the tax charged will be X rupees.

Example

- $X = 2, Y = 3, S = 1010110$

Initially there are 3 occurrences of 10 and 2 occurrences of 01; therefore, the tax charged will be $2 \cdot 2 + 3 \cdot 3 = 13$.

If we place all the $1's$ at the end of the string, the string S will become 0001111. Now the number of occurrences of 10 will become 0 and number of occurrences of 01 will become 1. Therefore, the tax charged will be $1 \cdot 2 + 0 \cdot 3 = 2$.

Minimum tax charged using **case 1** is Y rupees and minimum tax charged using **case 2** is X rupees. Since we can rearrange the string in any way we want, we will rearrange it according to the case which gives minimum tax charged. Therefore final output will be minimum of X and Y if string contains both $0's$ and $1's$.

TIME COMPLEXITY:

$O(1)$ for each test case.

PREREQUISITES:

bitwise XOR operation

PROBLEM:

Chef has a binary string S of length N. He wonders if it is possible to divide S into exactly K non-empty substrings such that each S_i belongs to exactly one substring and the XOR of each substring is the same. Can you help Chef to determine if it is possible to do so?

Note: XOR of substring $S_{L \cdots R}$ is defined as: $\text{XOR}(S_{L \cdots R}) = S_L \oplus S_{L+1} \oplus \cdots \oplus S_R$.

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

EXPLANATION:

The bitwise XOR of any binary string is either 0 or 1. Therefore if an answer exists then it is either K non-overlapping substrings having XOR equal to 1 or K non-overlapping substrings having XOR equal to 0. We can iterate over the binary string and check whether we can divide the string into K non-overlapping substrings having XOR equal to 0 or K non-overlapping substrings having XOR equal to 1
We can greedily select the first $K-1$ substrings having same $XOR(0\ or\ 1)$ and check whether there XOR is equal to the substring formed by the remaining characters. If it is equal output `YES` else `NO`.

We can show that if an answer exists it can always be constructed by this greedy approach.
Suppose there exists an answer such that the first $K-1$ substrings are not selected greedily, let it be $S[1, S_1], S[S_1+1, S_2], \ldots\ldots\ldots\ldots\ldots\ldots S[S_{K-1}+1, N]$, where S_i is the right end of the selected substring. Let there be and index $S_0 < S_1$ such that XOR of substring $S[1, S_1]$ is same as XOR of substring $S[1, S0]$, we select $S[1, S0]$ as the first substring and append the remaining characters of substring $S[1, S_1]$ to the start of the second string. The XOR of $1st$ and $2nd$ substrings remains same as $XOR\ of\ S[1, S_0] = S[1, S_1]$ and $XOR\ of\ S[S_0+1, S_2] = S[1, S_2] \oplus S[1, S_0] = S[S_1+1, S_2]$ as XOR of $S[1, S_1] = S[1, S_0]$ Now move to the next substring and keep on repeating the same procedure until all the $K-1$ substrings are selected greedily $i.e.$ we select the substring as soon as the XOR becomes equal to what we need $(0\ or\ 1)$.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

There are N sweets in the store. The cost of the i^{th} sweet is A_i rupees. Chef is a regular customer, so **after** buying the i^{th} sweet, he gets a **cashback** of B_i rupees.

Chef has R rupees. He is fond of all the sweets, so he wants you to calculate the maximum number of sweets he can buy. Note that he can buy the same type of sweet multiple times, as long as he has the money to do so.

EXPLANATION:

The cost of buying a sweet in this case is:

$$A[i] - B[i]$$

So what we can do is create another array of $A[i] - B[i]$ and sort it in increasing order. Then for each i, we would first check if the $A[i]$ value of the sweet is greater than the money we have left with us. If its greater than R then we cannot buy that sweet and we skip to the next one otherwise we can buy the sweet i. At this point the i_{th} sweet is also the cheapest sweet we can buy so we should buy as much of i_{th} sweet as possible. Thus we need to calculate for amount R, how many sweets of type i, we can buy. We can buy sweets till R is greater than or equal to $A[i]$.

$$sweets\_count = \frac{R - B[i]}{A[i] - B[i]}$$

we would add this to our answer and move on to the next sweet. Also we would reduce our R by $sweet\_count \times (A[i] - B[i])$, i.e

$$R = R - sweet\_count \times (A[i] - B[i])$$

TIME COMPLEXITY:

$O(NlogN)$ for each test case.

PROBLEM:

Given two strings A, B and Q queries, for each query we have to choose substring $A[L...R]$ and see if it is possible it to convert to $B[L..R]$ by cyclically incrementing two consecutive characters of A[L…R].

EXPLANATION:

Each query can be solved in O(R-L) as:
For query: L, R.
We will iterate over the substring from lowest to highest index and use the operation on $[i, i+1]$ to change $A[i] = B[i]$ for $i < R$.
If the last character $A[R]$ becomes equal to $B[R]$ after doing above operations the answer is YES else it is NO.

The above process of updating at each index and be mathematically expressed as:
Let $C_i = B_i - A_i$, be the number of operations to convert A_i to B_i if increasing single character were allowed.
Also for each query let's define D_i as the number of operation we apply at index i.
Then, mathematically $D_i = C_i - D_{i-1}$ as after applying D_{i-1} operation on index $i-1$, A_i changes to $A_i + D_{i-1}$, so to make it equal to B_i we do $B_i - (A_i + D_{i-1})$ operations on i^{th} index.

So, we get the value of D_R as:
$D_R = C_R - C_{R-1} + C_{R-2} + ...(-1)^{R-L} C_L$
As we cannot apply any operation at R^{th} position so if we want substrings to be equal D_R must be 0. Therefore the alternating sum of C_i in range L to R must be zero modulo 26 for answer to be YES .
Modulo 26 is required because operations are cyclic with length 26, as applying the same operation to an index 26 times, results in the same result. So, if $D_R = 26 \cdot p + t$ (where t is the remainder of D_R modulo 26), then applying operation at R, D_R times is same as applying operation at R, t times.

PROBLEM:

Given two integers A, B. You have to choose an integer X in the range $[minimum(A,B), maximum(A,B)]$ such that $\lceil \frac{B-X}{2} \rceil + \lceil \frac{X-A}{2} \rceil$ is maximum.

QUICK EXPLANATION

Try solving it for following 3 cases separately: 1) $A = B$, 2) $A > B$ and 3) $A < B$.

EXPLANATION

Let's divide the given problem in following three cases:

Case 1: $A = B$

In this case, we have only one choice for X that is $X = A = B$. In this case given sum is $\lceil \frac{B-X}{2} \rceil + \lceil \frac{X-A}{2} \rceil = \lceil 0 \rceil + \lceil 0 \rceil = 0$.

Case 2: $A < B$

Observation 1: Choose any X between the range $[A, B]$, the given expression would always be either $(B - A)/2$ or $(B - A)/2 + 1$.
Observation 2: Its always possible to achieve the sum $(B - A)/2 + 1$ if $A < B$.

- Subcase (i): $A - B$ is odd: in this case, by choosing $X = A$, we get the sum $\lceil \frac{B-A}{2} \rceil = (B - A)/2 + 1$.
- Subcase (ii): $A - B$ is even: in this case, by choosing $X = A + 1$, we get the sum $\lceil \frac{B-A-1}{2} \rceil + \lceil \frac{A+1-A}{2} \rceil = \frac{B-A}{2} + 1$ as $\lceil \frac{1}{2} \rceil = 1$.

Case 3: $A > B$

Lets solve it similar to the previous one.

- Subcase (i): $A - B$ is odd, we get the sum $(B - A)/2$ for every X.
- Subcase (ii): $A - B$ is even: in this case, by choosing $X = A - 1$, we get the sum $\lceil \frac{B-A+1}{2} \rceil + \lceil \frac{A-1-A}{2} \rceil = \frac{B-A}{2} + 1 + \lceil \frac{-1}{2} \rceil = \frac{B-A}{2} + 1$ as $\lceil \frac{-1}{2} \rceil = 0$.

TIME COMPLEXITY:

$O(1)$ per test case

PREREQUISITES:

Bitwise Operations

PROBLEM:

You are given an array A of size N and an integer X.

Find the count of all the pairs (i,j) such that $((A_i \oplus A_j)\ \&\ X) = 0$. Here $\oplus$ and $\&$ denote **bitwise XOR** and **bitwise AND** operations respectively.

EXPLANATION:

The given expression in the problem can be written as:

$$((A_i \oplus A_j)\&X) = (A[i]\&X) \oplus (A[j]\&X)$$

Also we know that:

$$a \oplus b = 0 \implies a = b$$

Thus we get:

$$(A[i]\&X) \oplus (A[j]\&X) = 0 \implies (A[i]\&X) = (A[j]\&X)$$

So in order to find the number of pair (i,j), we can construct a new array $A^{'}$ such that $A^{'}[i] = A[i]\&X, \forall i, 0 \leq i < n$ and then find the number of pairs (i,j) in this new array $A^{'}$ such that $A^{'}[i] = A^{'}[j]$ which can be done easily using a hash map to store the frequency of each element in $A^{'}$.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

Consider a $N \times N$ grid where N is odd. We are currently at point (x, y). We need to find the minimum cost to reach the center of the grid by using some operations where in an operation we can move from (i,j) to $(i+1,j+1)$, $(i+1,j-1)$, $(i-1,j+1)$ or $(i-1,j-1)$ with cost 0 and we can move from (i,j) to $(i,j+1)$ or $(i,j-1)$ with cost 1.

EXPLANATION:

- Let $C = \frac{N+1}{2}$. The center of the grid is (C, C).
- Without losss of generality, let us assume $i < C$ and $j < C$. (The similar casework applies for remaining cases also.)
- I claim that the answer will be always 0 or 1.
- Let $p = C - X$ and $q = C - y$. If $p = q$, we can apply the operation $(i,j) \rightarrow (i+1,j+1)$ p times and reach the center with 0 cost.
- If $p < q$, and $q - p$ is **odd**, we can apply the operation $(i,j) \rightarrow (i+1,j+1)$ p times, then apply the pair of operations $(i,j) \rightarrow (i+1,j+1) \rightarrow (i,j+2)$ $\frac{q-p-1}{2}$ times. Now we will end up at $(C, C-1)$ with 0 cost. With a cost of 1, we can move from $(C, C-1)$ to (C, C).
- If $p < q$, and $q - p$ is **even**, we can apply the operation $(i,j) \rightarrow (i+1,j+1)$ p times, then apply the pair of operations $(i,j) \rightarrow (i+1,j+1) \rightarrow (i,j+2)$ $\frac{q-p}{2}$ times. Now we will end up at (C, C) with 0 cost.
- Similarly, we can check for all the cases. Therefore, the final answer is 0 if the parity of $q - p$ is 0, else the answer is 1.

TIME COMPLEXITY:

$O(1)$ for each test case.

PREREQUISITES:

Sieve of Eratosthenes

PROBLEM:

Chef needs to go from X to Y. For this he can consider the following two operations:

Suppose chef is at location C. Now he can

- Go from C to $C+1$ in one step
- Go from C to $C+2$ in one step if $C+2$ is prime.

We need to find the minimum number of steps to go from X to Y.

EXPLANATION:

- If the move C to $C+2$ is possible, we must always prefer this rather than going from $C \rightarrow C+1 \rightarrow C+2$ because it always saves us one step.
- Let us find the number of prime numbers $P \leq Y$ for which $P-2 \geq X$. Let this value be tot.
- Whenever we want to go to such P we will prefer the second operation, i.e going from $P-2$ to P directly using one step.
- Therefore, our final answer will be $Y-X-tot$.
- We can precompute the number of primes from 1 to N using the sieve method. From this information, we can easily calculate the number of primes in a range in $O(1)$ time.

TIME COMPLEXITY:

$O(N \log \log N)$ where $N = 10^7$ for performing sieve.

PREREQUISITES:

Fenwick Trees (Binary Indexed Trees), Segment Trees, Range Updates

PROBLEM:

Given an array A consisting of size N. Process Q queries of the following types:

$1\ L\ R\ X$ - Add $(X+i-L)^2$ to A_i, for all i in $[L,R]$

$2\ i$ - Output A_i for $i \in [1,N]$

QUICK EXPLANATION:

The given queries can be easily processed with binary indexed tress or segment trees.

EXPLANATION:

Processing type 1 queries

Let's look at the first query:
$1\ L\ R\ X$ - Add $(X+i-L)^2$ to A_i, for all i in $[L,R]$
Here we are adding $(X+i-L)^2$. Let $y = X-L$. Now lets expand this term:
$(X+i-L)^2 = (X-L+i)^2$
$= (y+i)^2$
$= y^2 + 2*y*i + i^2$

Now lets say we performed two queries of type 1 with $y = y_1$ and $y = y_2$ respectively. Let's see what value we added to A_i after the two queries. We add following to A_i.
$(y_1^2 + 2*y_1*i + i^2) + (y_2^2 + 2*y_2*i + i^2)$
$= (y_1^2 + y_2^2) + (2*y_1 + 2*y_2)*i + (1+1)*i^2$

Similarly after we perform k queries of type 1, A_i will be increased by:
$(y_1^2 + y_2^2 + ... + y_k^2) + (2*y_1 + 2*y_2 + ... + 2*y_k)*i + (1+1+...+1(k\ times))*i^2$

If we can maintain following sums $\sum y^2$, $\sum 2*y$ and $\sum 1$ for each i then we can easily answer what would be the value of A_i after performing these queries. These values can be maintained by using three binary indexed trees or segment trees. Let the data structures B_1 maintain the sum $\sum y^2$, B_2 maintain the sum $\sum 2*y$ and B_3 maintain the sum $\sum 1$.

Now we can break the query of type 1 in following steps:

- Add y^2 to the range $[L,R]$ in B_1.
- Add $2*y$ to the range $[L,R]$ in B_2.
- Add 1 to the range $[L,R]$ in B_3.

These are simple update queries which can be performed in $O(logn)$ time in both segment trees and binary indexed trees.

Answering type 2 queries

As we saw in the previous section, after k queries of type 1, A_i will be increased by:
$(y_1^2 + y_2^2 + ... + y_k^2) + (2*y_1 + 2*y_2 + ... + 2*y_k)*i + (1+1+...+1(k\ times))*i^2$

$= \sum y^2 + i * \sum 2 * y + i^2 * \sum 1$

Let the point query $q_1(i)$ return the value at index i in B_1, $q_2(i)$ return the value at index i in B_2 and $q_3(i)$ return the value at index i in B_3. These are simple get queries for the value at index i which can be easily processed in $O(logn)$ time in both segment trees and binary indexed trees.

Now the value added at index i would be:
$q_1(i) + i * q_2(i) + i^2 * q_3(i)$

So answer to the query 2 i would be $A_i + q_1(i) + i * q_2(i) + i^2 * q_3(i)$.

Algorithm

- Keep 3 fenwick arrays or segment trees B_1, B_2, B_3.
- For queries of type 1
  - Now let $y = X - L$.
  - For B_1 do a range update: add y^2 for all i from L to R.
  - For B_2 do a range update: add $2 * y$ for all i from L to R.
  - For B_3 do a range update: add 1 for all i from L to R.
- For queries of type 2
  - Let point query in i^{th} DS on index j be represented by $q_i(j)$
  - Answer to the query is $A_i + q_1(i) + i * q_2(i) + i^2 * q_3(i)$.

TIME COMPLEXITY:

$O(QlogN)$ for processing Q queries as each query takes $O(logn)$ time

PREREQUISITES:

Prime factorization

PROBLEM:

There is rectangular screen in coordinate plane from $(0, 0)$ to (N, M). We need to divide the rectangle into vertical columns with equal size with borders as integer coordinates. Then each column is further divided into windows with the following properties:

- For a column, the size of each window must be same and must have borders as integer coordinates.
- There must be atleast two windows in a single column.
- No two windows in **different columns** have the same y coordinate for horizontal edges, except for the borders of the screen.

We need to output the maximum number of columns that can be created.

EXPLANATION:

- Let there be a total of tot columns in the final answer with the number of windows in each column as $c_1, c_2 \dots c_{tot}$.
- To ensure the horizontal edges of two windows of different columns i and j have different y coordinates, we need to have $gcd(c_i, c_j) = 1$.
- We can easily prove this by contradiction: Let $gcd(c_i, c_j) = k > 1$. Then, $c_i = k \cdot p$ and $c_j = k \cdot q$ for some positive integers p and q. Now, window in column i has size $s_i = \frac{M}{k \cdot p}$ and window in column j has size $s_j = \frac{M}{k \cdot q}$. It can be observed clearly that p^{th} window from the bottom in column i has a horizontal edge which will have the same y coordinate the q^{th} window from the bottom in column j. In other words, $p \cdot s_i = q \cdot s_j = \frac{M}{k} < M$. We arrived at a contradiction because our assumption is wrong. Therefore, $gcd(c_i, c_j)$ must be equal to 1.
- Now our main goal is to maximize the value of tot. This can be realised by first doing the prime factorization of M. This can be done in $O(\sqrt{M})$ by the classic prime factorization method. Let there be a total of y primes and let $M = p_1^{x_1} \cdot p_2^{x_2} \dots p_y^{x_y}$.
- The maximum value of tot we can have is y with possible $c_1 = p_1, c_2 = p_2 \dots c_y = p_y$. Why? Suppose $tot > y$. Then atleast two columns i and j will have $gcd(c_i, c_j) \neq 1$. This is because of **pigeon-hole principle**. If $tot > y$ and we only have y primes available and every column i must have $c_i > 1$, atleast one prime must be repeated in two columns which results in their gcd greater than 1. Therefore, $tot = y$ is the maximum possible.
- We are now left to find the maximum number less than tot which is divisible by N for the borders of columns to have integer coordinates. This can be simply done by iterating over i from 1 to tot and choosing maximum i for which N is divisible by i.

TIME COMPLEXITY:

$O(\sqrt{M})$ for each test case

PREREQUISITES:

Observation, Greedy, Median

PROBLEM:

Given a city with n houses. Each house is located at a distinct **positive integer** coordinate $a_1, a_2, \dots a_n$. The chef is planning to create k hills in the city. Note that two hills cannot be present at the same location but the location of hills and houses can overlap and each hill will also be at any **real integer** coordinate.
Each citizen would want to travel to the farthest hill from his house. Help the Chef find the minimum sum of distance traveled by everyone.

QUICK EXPLANATION:

The distance sum is minimum when the median of the hills is same as median of the houses and only the starting position of hills matter because the hills should be located with 1 unit distance to minimize the distance sum.

EXPLANATION:

Observations

1. The distance sum is minimum when the median of the hills is same as median of the houses.
2. Also we should place all hills nearby median hence placing them at unit distance is best choice.
3. So only starting position of hills matters, other hills follow it by unit distances.

Solution

Now there can be two possible medians for the houses, which would be $a[n/2]$ and $a[(n-1)/2]$. As we will place the hills at unit distances and want to make the medians same, we should match the left median of houses with left median of hills and right median of the houses with right median of the hills.
Hence for the median $a[n/2]$, we will start placing the hills from $a[n/2]-k/2$ and for the median $a[(n-1)/2]$, we will start placing the hills from $a[(n-1)/2]-(k-1)/2$.

Fun Fact: Both medians are always symmetric and both of them gives same answer.

Also if first hill is at position x, the last hill will be at position $x+k-1$ considering we are putting all hills continuously at unit distances. So i-th house citizen will travel $D_i = max(abs(x-a[i]), abs(x+k-1-a[i])$ distance to visit farthest hill.

Finally find sum of distances for any of two possible starting hill choices $a[n/2]-k/2$ and $a[(n-1)/2]-(k-1)/2$. As answer is always same for both choices.

Why only median choices are optimal

Consider all 4 cases and try to prove them —

1. n and k both even
2. n even and k odd
3. n odd and k even
4. n and k both odd

This way we can easily prove that, in all 4 cases, we will get the two median choices as $a[n/2]$ and $a[(n-1)/2]$.

TIME COMPLEXITY:

$O(n)$ per test case

PREREQUISITES:

Graph, BFS/Level Order Traversal, Multiset
(Alternatively Dijkstra's Shortest Path Algorithm)

PROBLEM:

Chef's college is starting next week. There are S subjects in total represented using an array A, and he needs to choose K of them to attend each day, to fulfill the required number of credits to pass the semester. There are $N + 1$ buildings. His hostel is in building number 0. Subject i is taught in building $S[i]$. After each subject, there is a break, during which he goes back to his hostel. There are M bidirectional paths of size 1 which connects building u to building v. Find the minimum possible total distance Chef needs to travel each day if he chooses his subjects wisely.

Note that further in the editorial, the term level of a node x with respect to a fixed node in a graph has been used to describe the number of edges in the smallest path from the fixed node to node x.

QUICK EXPLANATION:

As the hostel and other buildings form a graph with unit weights of all paths, we only need to know the levels of this graph in which each of the **considered** buildings lie (**considered** buildings being the ones teaching one of the subjects from 1 to S inclusive).

Considering the hostel to be the lone source at level 0, the one way journey to any building will be x units from the hostel, where x represents the level in which the target building lies. Selecting the K subjects which have the nearest buildings assigned to their teaching will result in the minimum possible distance Chef needs to travel.

EXPLANATION:

The college has been given to have a hostel building and N remaining buildings where subjects are taught. Which of these buildings are connected by paths of 1 unit length is also given. This data given can be represented as a graph with buildings as nodes and paths as edges. Since all the paths are of unit length, the weight of each edge will be assigned as 1. An implication of unit long paths is that the distance traveled from the hostel to building B will be equal to the level of B with respect to the hostel in the graph.

Proof

Consider the hostel (node 0) to be at level 0, all nodes that can be reached from node 0 by means of a single edge are at level 1 with respect to node 0. Similarly any building reachable from the hostel by means of x bidirectional paths each connecting a unique pair of buildings will be at level x with respect to the hostel.

We establish the relation between the number of edges e (needed to be crossed from hostel to reach any other building) and level L (of a building with respect to the hostel):

$$e = L$$

If we need to travel from 0 to x in this graph and e number of edges E_1 to E_e constitute the path from 0 to x, then the total distance D of our path will be given by:

$$D = \sum_{i=1}^{e} d(E_i)$$

where $d(E_i)$ represents the length of edge E_i. In this case, the lengths of individual paths are given to be 1 each, thus we obtain:

$$D = \sum_{i=1}^{e} 1$$
$$D = e$$

which from the earlier established relation can be written as:

$$D = L$$

Where L is the level of node x with respect to node 0.

As the graph is not acyclic, a single node x can have multiple ways of reaching from 0 through different sets of edges. It may seem to belong to different levels due to this, in this case we will consider that it belongs to the smallest of these levels (closest one to level 0) because we are required to find the minimum distance traveled by Chef implying if more than a single path exist to a node from 0 we shall follow the smallest weighted one.

Example

If there exist 2 paths from node 0 to x consisting of e and f edges respectively ($e < f$). Let the paths be constituted by edges E_1 to E_e and F_1 to F_f respectively. In such a case, our level L of node x with respect to node 0 will be e enabling us to have smaller path covered from hostel to building x by Chef.

In case $e = f$, both paths will result in the same level of x, thus no conflict arises.

To implement this assignment of levels to nodes, we can use breadth first traversal (level order traversal) of the graph we have converted the input into. More about this algorithm can be found **here** and **here**.

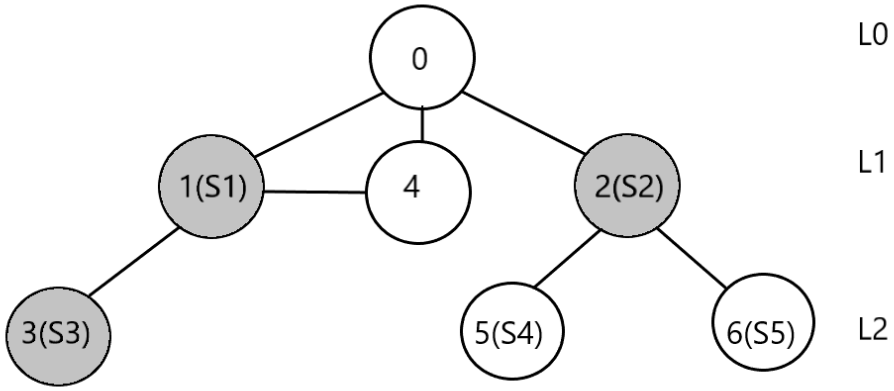
Starting the level order traversal from node 0 and level 0, as we encounter that one whole level has been popped out of the queue and their children (the next level) have been pushed into it, we increment the level and continue the traversal.

Once all the buildings in which 1 to S subjects are taught have been given a level with respect to the hostel, we need to select K from among these such that they hold the minimum levels. The buildings can appear more than once i.e. there can exist cases whereby more than 1 subject is being taught in a single building, thus while selecting the K buildings with minimum levels we need to make sure the same building can be selected as many times as the number of different subjects being taught in it.

To establish this we utilize a multiset. Traverse all buildings in which 1 to S subjects are being taught and insert their corresponding levels into a multiset. The sum of first K elements of this multiset will give us the distance traveled by Chef from the hostel to those buildings in which the subjects he chose are being taught, thus we multiply this by 2 to obtain the final answer (as he returns to his hostel after each subject, thus needing to travel the same distance he had to while coming from hostel to the building).

The sample input provided with the problem statement would have a graph as represented below:

```
6 7 5 3
0 1
0 2
0 4
1 3
1 4
2 5
2 6
1 2 3 5 6
```



Where L_i represents the i_{th} level, S_i represents the i_{th} subject, and the numbers written in the nodes represent the building numbers, 0 being the hostel. The colored nodes are selected for obtaining minimum traveled distance.

Alternatively, the buildings 5 or 6 could also have been chosen instead of 3, i.e. subject $S4$ or $S5$ instead of $S3$ would also have given the minimum distance (8 in this case). This is because all the alternatives suggested lie in the same level as the selected node and thus if selected in place of the selected node will not hinder the minimum distance.

TIME COMPLEXITY:

$O(max(N, log(S) \times S))$ per test case.

Because the level order traversal takes $O(N)$ time and the multiset construction takes $O(log(S) \times S)$ i.e. S insert operations of $O(s_i)$ each, for all s_i from 0 to $S-1$.

Complexities for operations on used containers can be found - **for multiset** and **for queue**.

PROBLEM:

Chef considers a string consisting of lowercase English alphabet *beautiful* if **all** the characters of the string are **vowels**.

Chef has a string S consisting of lowercase English alphabet, of length N. He wants to convert S into a *beautiful* string T. In order to do so, Chef does the following operation on **every** character of the string:

- If the character is a **consonant**, change the character to its **closest vowel**.
- If the character is a **vowel**, don't change it.

Chef realizes that the final string T is not unique. Chef wonders, what is the total number of **distinct** *beautiful* strings T that can be obtained by performing the given operations on the string S.

Since the answer can be huge, print it modulo $10^9 + 7$.

Note:

- There are 26 characters in the English alphabet. Five of these characters are vowels: `a`, `e`, `i`, `o`, and `u`. The remaining 21 characters are consonants.
- The closest vowel to a consonant is the vowel that is at least distant from that consonant. For example, the distance between the characters `d` and `e` is 1 while the distance between the characters `d` and `a` is 3.
- The distance between the characters `z` and `a` is 25 and **not** 1.

EXPLANATION:

In this problem we need to change all the consonants to their nearest vowel. The consonants that have a unique vowel nearest to it will just have one choice, that is to change into that vowel, however for vowels that have 2 vowels at same closest distance, it will have 2 choices. Thus we just need to calculate all consonants that have 2 choices. Let us denote this by $count$. Then our final answer would be:

$$answer = 2^{count}$$

TIME COMPLEXITY:

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

PROBLEM:

An integer x is said to be a **Perfect Power** if there exists **positive** integers a and b (i.e a, and b should be ≥ 1) such that $x = a^{b+1}$.

Given an integer n, find the closest Perfect Power to n. If there are multiple *Perfect Powers* equidistant from n and as close as possible to n, find the smallest one.

More formally, find the smallest integer x such that x is a Perfect Power and $|n - x| \leq |n - y|$ over all Perfect Powers y.

EXPLANATION:

Have a close look at the constraints of N, it's just $1e18$. Why just man, it's so big?

Remember we will be dealing with the powers and it will take few iterations to get to $1e18$ even if we start will the lowest base possible $i.e\ 2$. It just takes 60 iterations to reach a value of $1e18$.

Now let's look at how we can solve this problem. One of the basic ideas is to iterate for all bases and their powers. The problem with this approach is that the numbers that can act as bases are very large. A number as large as $1e9$ can act as a base.

Can we optimize it?

For a moment change the condition for power and say every power should be greater than 3. Then what is the maximum value that base (a) can take, is it less than $1e5$. Hence we can surely iterate over the bases here if the power allowed to us was greater than 3.

Now the only powers that we are not considering and are in problem are squares and cubes. But for any number, it is quite easy to check a perfect square or a perfect cube closest to it.

But yes, there is another problem with test cases, these test cases are so large that even after optimizations, we will end up hitting TLE. One of the ways is pre-computation of all the perfect powers and then apply binary search on it to find the closest power.

TIME COMPLEXITY:

$O(N * log(N))$, where $N \approx 1e5$

PREREQUISITES:

Combinatorics, Graph Theory

PROBLEM:

You are given an array A of length N such that $1 \leq A_i \leq N$ and an integer M. Count the number of arrays B of length N, such that

- $B_i \neq B_{A_i}$
- $1 \leq B_i \leq M$

Since the answer can be large, print it modulo $10^9 + 7$.

QUICK EXPLANATION:

- Build a graph with N edges as (i, A_i). The graph will have some connected components.
- Each connected component has exactly one cycle.
- Calculate the answer for the cycle. Each non-cycle node can be filled in $(M - 1)$ ways. The total answer for a component is the product of the answer for cycle as well as that of non-cycle nodes.
- Final answer is the product of the answers of all connected components.

EXPLANATION:

Observation

We are given that $B_i \neq B_{A_i}$. To connect the i^{th} position with the $A_i{}^{th}$ position, we can build a graph with N edges. Each edge is generated using the vertices i and A_i.

Considering $1 \leq A_i \leq N$, the graph would consist of various connected components. Each of the connected components will have exactly one cycle. In other words, the graph would be a tree with one extra edge which forms a cycle.

Proof

Any connected component of size K will have exactly K edges. For more than one cycle to exist, there has to be atleast $K + 1$ edges. Also, for no cycles, there can't be more than $K - 1$ edges. Thus, there is exactly one cycle present.

For each connected component, let us look at this cycle part separately.

Subproblem

Problem: Find number of arrays B of length $N(\geq 2)$ such that:

- $B_1 \neq B_N$
- $B_i \neq B_{i+1}$ $(1 \leq i < N)$
- $1 \leq B_i \leq M$

Solution: We need to find the number of arrays such that no two consecutive elements have the same value (including first and last element) and each element has value in range $[1, M]$. We can use dynamic programming to calculate this.

Let $dp[i][0]$ denote the number of ways to fill first i elements such that $B_j \neq B_{j+1}$ $(1 \leq j < i)$ and $B_1 \neq B_i$.
Similarly, let $dp[i][1]$ denote the number of ways to fill first i elements such that such that $B_j \neq B_{j+1}$ $(1 \leq j < i)$ and $B_1 = B_i$.
The answer to our problem is the value $dp[N][0]$.

- **Base Case:** $dp[1][0] = 0$, there is no way in which we can fill the first element such that it is not equal to the first element. Similarly, $dp[1][1] = M$.
- **Calculating** $dp[i][1]$**:** Since there is only one way of filling the i^{th} element, $dp[i][1]$ is nothing but $dp[i-1][0]$.
- **Calculating** $dp[i][0]$**:** If the $(i-1)^{th}$ element is equal to the first element, the i^{th} element can be filled in $(M-1)$ ways (all numbers except the value of the first element). Else, there are $(M-2)$ ways to fill that (all numbers except the value of the first and the $(i-1)^{th}$ element). Thus, $dp[i][0] = (dp[i-1][1] \cdot (M-1) + dp[i][0] \cdot (M-2))\%mod$.

Conclusion

Consider a connected component with X nodes. Let this component contain a cycle of length Y. For the cycle part, the answer is $dp[Y][0]$. For all other nodes, which are not part of the cycle, there are $(M-1)$ ways to fill each one of them. Thus, the answer for this component is $(dp[Y][0] \cdot (Y-X)^{M-1})\%mod$.

The final answer would be the product of the answers of all such components.

TIME COMPLEXITY:

The time complexity is $O(N)$ per test case.

PREREQUISITES:

You should be familiar with loops and the concept of **Greatest Common Divisor**.

PROBLEM:

Given an integer N, find the number of tuples (w, x, y, z) such that $1 \leq w, x, y, z \leq N$ and $\frac{w}{x} + \frac{y}{z}$ is an integer.

For example, if $N = 2$, the tuple $(1, 2, 1, 2)$ satisfies both conditions, i.e. $\frac{1}{2} + \frac{1}{2} = 1$ is an integer, however $(1, 1, 1, 2)$ does not since $\frac{1}{1} + \frac{1}{2} = 1.5$ is not an integer.

QUICK EXPLANATION

- Let us fix the values of w and x such that $gcd(w, x) = 1$. This will constraint the values z can take, and we can loop over the possible values of z to get the final answer.
- What if w and x are not co-prime? We can consider all these pairs while looping over the above pairs of (w, x).

EXPLANATION:

The constraint of the problem suggests that we can fix 2 variables and then try to calculate the answer. Let us try to fix both w and x, and try to proceed further. We can fix other pairs also, like y and z, and get the answer.

So let us proceed by fixing w and x. Also, let w and x be co-prime. If not, they can be reduced to $w^{'}$ and $x^{'}$, such that $w^{'}$ and $x^{'}$ are co-prime and $\frac{w}{x} = \frac{w^{'}}{x^{'}}$. Hence, their contribution can be calculated when dealing with $(w^{'}, x^{'})$.

Now, we have $gcd(w, x) = 1$. So, for $\frac{w}{x} + \frac{y}{z}$ to be an integer, z needs to be a multiple of x. Let us say that $z = c \cdot x$. Let us iterate over all the possible values of z.

So our problem becomes:

$\frac{w}{x} + \frac{y}{c \cdot x}$ is an integer, which can be written as $\frac{c \cdot w + y}{c \cdot x}$ is an integer.

Note that we know the values of $c \cdot w$ and $c \cdot x$. The minimum value of y that satisfies the above equation will be $y^{'} = c \cdot x - (c \cdot w)\%(c \cdot x)$. The general value of y that will satisfy this equation can be represented as $y = y^{'} + \lambda \cdot (c \cdot x)$, with the constraint that $y \leq N$. We can calculate the number of valid $y^{'}s$ as $\frac{N - y^{'}}{z} + 1$.

Also note that this pair of (w, x) represents some other pairs which can be reduced to the form $\frac{w}{x}$. Number of such pairs will be $min(N/w, N/x)$.

TIME COMPLEXITY:

$O(N^2 \cdot \log N)$ for each test case.

PROBLEM

Given duration of shows of S seasons, each season having an intro song, Determine the time taken to watch all the episodes, while skipping intro song in each episode except once every season.

QUICK EXPLANATION

Simply subtract i-th intro song duration $(E_i - 1)$ times, as i-th song would be skipped $E_i - 1$ times. Then take the total of all the episode times.

EXPLANATION

Since this is a simple problem, I'll explain the solution from two viewpoints for interested readers. It's perfectly okay to choose any veiwpoint as feels comfortable. Both are essentially same in idea.

Viewpoint 1

In this view point, we would compute the time spent on each episode. We'd subtract the duration of intro song from all except first episode for all seasons. So we have exact times spent on each episodes. We can take total of these and determine total duration.

Viewpoint 2

In this viewpoint, we'd focus on computing the time spent viewing the intro song and time spent viewing content in episodes. Since each intro song is watched exactly once, so sum of duration of each intro song determines the total time spent hearing intro songs.

Computing the time spent viewing content is subtracting the duration of intro song from each episode and adding the remaining episode time.

TIME COMPLEXITY

The time complexity is $O(\sum E_i)$ per test case.
The space complexity is $O(1)$ per test case.

PREREQUISITES:

Two Pointers, Implementation

PROBLEM:

You are given two arrays - $F_1, F_2, \ldots, F_n$, which denote the times when an important event happens in the football match. And similarly $C_1, C_2, \ldots, C_m$ for cricket.

You sadly have the remote in hand. You start out by watching the El Clasico. But whenever an Important event happens in the sport which isn't on the TV right now, you will be forced to switch to that sport's channel, and this continues, i.e., you switch channels if and only if when an important event in the other sport happens.

Find the total number of times you will have to switch the channels.

EXPLANATION:

We need to find the total number of times we have to switch the channel. A channel is switched whenever an important event happens in the sport which isn't on the TV right now. We can implement it in many ways, one of the ways is to use the Two Pointer Approach.

Case 1: We are watching Football currently

Suppose we are at i^{th} index of Football event and at j^{th} index of Cricket Event. Since it is already mentioned in the constraints that $F_i \neq C_j$ for any i and j. Hence there are two possibilities left.

- If $F_i < C_j$. It means that Football has some important event before the Cricket Event. Since currently, we are watching Football hence we need not switch the channel. We will increment i and repeat the above procedure again.
- If $C_j < F_i$. It means that Cricket has some important event before the Football Event. Since currently, we are watching Football hence we need to switch the channel. We will increment j and now we will be watching Cricket.

Case 2: We are watching Cricket currently

- If $F_i < C_j$. It means that Football has some important event before the Cricket Event. Since currently, we are watching Cricket hence we need to switch the channel. We will increment i and now we will be watching Football.
- If $C_j < F_i$. It means that Cricket has some important event before the Football Event. Since currently, we are watching Cricket hence we need not switch the channel. We will increment j and repeat the same procedure again.

What happens when all the important events of one of the sport are completed ?

In this case, we will be watching those sports currently and since the important events are about to come for the other sports hence we are forced to switch the channel. However, once we switched the channel we need to switch the channel again as all the important events of other sport are completed.

By following the above approach we will be find the number of times we need to switch the channel and finally we will output this number.

TIME COMPLEXITY:

$O(N + M)$ per test case

PREREQUISITES:

Dijkstra

PROBLEM:

There are N cities and M bidirectional roads. Each road has a particular cost to travel on.

K cities among these have corona testing facility for a particular amount (different cities may have different prices for testing).

Determine for each city, the minimum cost to travel to any corona testing facility and get tested.

EXPLANATION:

Model the problem as a weighted graph with $N + 1$ nodes. Draw an edge between node 0 and node i with a corona testing facility (for all i), and give it weight C_i.

Now, we have reduced the problem to finding the shortest path from each city to node 0. This is equivalent to finding the shortest path from node 0 to each city, which is a classical Dijkstra problem!

TIME COMPLEXITY:

Dijkstra (with minimum priority queue) takes

$$O(V + E \log V)$$

which is the time complexity per test case.

PROBLEM:

Chef knows two languages spoken in Chefland but isn't proficient in any of them. The first language contains characters $[a - m]$ and the second language contains characters $[N - Z]$.

Given K words of the sentence as $S_1, S_2, \ldots, S_K$ tell whether it is a possible sentence framed by Chef, i.e, it contains only the characters from the two given languages and each word contains characters from a single language.

EXPLANATION:

We are given K words of the sentence, we just need to check whether each word belongs to exactly one language.

A word belongs to the first language if all the characters of the word belong to $[a - m]$, and it belongs to the second language if all the characters of the word belong to $[N - Z]$. If a word contains the characters of both the languages or some characters are neither in language first nor in second, then the sentence is not valid.

Pseudo Code to check the word

```
// s is some word of a sentence

for(auto ch: s)
{
      if(ch>='a' && ch<='m')
        fst=false;
      else if(ch>='N' && ch<='Z')
        snd=false;
      else
        ok=false;
}

if(!fst && !snd)
    ok=false;
```

TIME COMPLEXITY:

$O(|S|)$ per test case

where $|S|$ is the length of sentence

PREREQUISITES:

Math, Bit-wise Operations

PROBLEM:

Given an array A and B of lengths N.
$H(i,j) = i \cdot j - D \cdot (B_i \bigoplus B_j)$
Find the Maximum value of $H(i,j)$ such that $A_i \leq j$

QUICK EXPLANATION:

It can be proved that only i,j such that $N - 2 * D \leq i,j$ can contribute to the answer.
Hence we can use brute-forced on this range having time complexity $O(D^2)$

EXPLANATION:

It can be observed that for every $i \in N$, $A_i \leq N$, Hence Every i can have a reaction with N.

$H(N-1,N) = (N-1) \cdot N - D \cdot (B_{N-1} \bigoplus B_N)$
The Maximum value $(B_{N-1} \bigoplus B_N)$ can have is $2 \cdot N$
Hence, $H_{\min}(N-1,N)$ is $(N-1) \cdot N - 2 \cdot D \cdot N$

Now, Let i ∈ $[1, N-1]$
$H(i,N) = i \cdot N - D \cdot (B_i \bigoplus B_N)$
The Minimum value of $(B_i \bigoplus B_N)$ is 0
Hence ,$H_{\max}(i,N) = i \cdot N$
also $H_{\max}(i,j) < H_{\max}(i,N) \forall j \in [2, N-1]$
So $H(i,j)$ is at most $H_{\max}(i,N) = i \cdot N$

Any i can possibly be answer only if
$H_{\min}(N-1,N) < H_{\max}(i,N)$
$\Rightarrow (N-1) \cdot N - 2 \cdot D \cdot N < i \cdot N$
$\Rightarrow (N-1) - 2 \cdot D < i$
$\Rightarrow N - 2 \cdot D \leq i$

So we can check $\forall i,j \in [N - 2 \cdot D, N]$ and $i < j$

PREREQUISITES:

Dynamic Programming

PROBLEM:

Consider an array A with N positive integers.

Let B be defined as an array of N integers such that for each $1 \leq i \leq N$:

- $B_i = A_i - 1$, if A_i is even.
- $B_i = A_i + 1$, if A_i is odd.

You are given an array C of size $2 \cdot N$ which is formed by taking all the elements of the arrays A and B and rearranging them.
You need to find the number of distinct arrays A whose sum of elements is K, from which C can be obtained.

Note: Two arrays X and Y are considered to be distinct if there exists no rearrangement of X which is the same as Y.

Since the final answer can be very large, print the answer modulo $10^9 + 7$.

EXPLANATION:

Some Observations:

- If an odd number x occurs m times in array C, then $x + 1$ should also occur m times, (otherwise the answer will be zero), because if x occurred in array A, $x + 1$ must occur in array B and vice versa.
- Similarly an even number y occurs m times in array C, then $y - 1$ should also occur m times in array C
- Number of odd elements is equal to Number of even elements = N.
- Minimum possible sum of array A will be the sum of all the odd numbers of array C
- Maximum possible sum of array A will be the sum of all the even numbers of array C.

This implies that if $K <$ sum of all odd numbers in array C OR $K >$ sum of all even numbers in array C, the answer is zero.
Now assume D = $K-$ sum of all odd numbers in array C.
Consider a possible array A with all the odd elements of C. We need to select any D even elements from C and replace each of them with their adjacent smaller odd number to increase the sum by D.
We denote the number of distinct even numbers in array C by M and the count of each even number by $P_1, P_2, ..., P_m$. Thus we need to find the solution of the equation $X_1 + X_2 + ... + X_m = D$, such that $0 \leq X_1 \leq P_1, 0 \leq X_2 \leq P_2, ...0 \leq X_m \leq P_m$.

The solution to the above equation can be obtained by standard dynamic programming under given constraints.

TIME COMPLEXITY:

$O(N^2)$ for each test case.

PREREQUISITES

Dynamic Programming, Combinatorics

PROBLEM

Consider an array A of length N where $0 \leq A_i \leq 10^5$. Construct an array B of length $N-1$ as $B_i = min(A_i, A_{i+1})$.

You are given the array B, find the number of arrays A of length N, which can be used to construct array B by the above process.

EXPLANATION

This is a DP combinatorics problem, So I recommend pen and paper as well. Also, using $M = 10^5$ throughout the editorial.

What we can see here is that, Either $B_i = A_i$ or $B_i = A_{i+1}$. Let's consider first element A_1.
Either

- $A_1 > B_1$ OR
  In this case, it doesn't matter which value A_N takes, as long as $B_1 < A_1 \leq M$, So there are $(M - B_1)$ ways to choose the last value. Also, A_2 must be equal to B_1 in this case.
- $A_1 = B_1$
  In this case, A_2 can take any value in range $[max(B_1, B_2), M]$.

We can see that we only care about whether $B_i = A_i$ or $B_i = A_{i+1}$ and not the actual values of A_i and A_{i+1}.

This suggests a kind of DP where we maintain one state for keeping track of index, and a flag to determine whether $B_i = A_i$ or $B_i = A_{i+1}$

Let's denote $f(i)$ as the number of ways to select first i elements of A in valid way such that $A_i = B_i$
Also, let's denote $g(i)$ as the number of ways to select first $i+1$ elements of A such that $A_{i+1} = B_i$ and $A_i > A_{i+1}$. This second condition is added, to avoid double counting of the case where $A_i = A_{i+1} = B_i$

The final answer would be $f(N-1) \times (M - B_{N-1} + 1) + g(N-1)$, as this covers both the cases when $A_{N-1} = B_{N-1}$ (first term) and $A_{N-1} > B_{N-1}$ (second term).

For simplicity, let's denote $w(x) = M - x + 1$, we we need to use it quite frequently. Required answer becomes $f(N-1) * w(B_{N-1}) + g(N-1)$. $w(B_{N-1})$ denotes the number of ways to select last element when first $N-1$ elements are already chosen.

Now, let's see how base cases and transitions work.

Base cases

We can see that $f(1) = 1$, as $A_1 = B_1$ is the only way.
For $g(1)$, we choose $A_2 = B_1$ and $A_1 > B_1$ which can be done in $(M - B_1) = w(B_1 + 1)$ ways.

Hence, $f(1) = 1$ and $g(1) = w(B_1 + 1)$ are the base cases.

Transitions

We need to find formula for $f(i)$ and $g(i)$ represented by terms $f(i-1)$ and $g(i-1)$.

For the computation of $f(i)$, the following cases need to be considered.

- Contribution of $f(i-1)$ in $f(i)$.
  If we have $B_i \geq B_{i-1}$, then we can choose $A_i = B_i$, which can be done in $f(i-1)$ ways.
- Contribution of $g(i-1)$ in $f(i)$
  If an array A is included in both $f(i)$ and $g(i-1)$, it implies $A_i = B_i$ and $A_i = B_{i-1}$. This can only happen when $B_i = B_{i-1}$. Hence, if $B_i = B_{i-1}$, each way included in $g(i-1)$ contributes one way in $f(i).

Similarly, for computation of $g(i)$, the following cases need to be considered.

- Contribution of $f(i-1)$ in $g(i)$
  $f(i-1) sets $A_{i-1} = B_{i-1}$ and $g(i)$ sets $A_{i+1} = B_i$. So we only need $A_i \geq max(B_{i-1}, B_i + 1)$. Hence, each way in $f(i-1)$ contributes $w(max(B_{i-1}, B_i + 1)$ ways in $g(i)$.
- Contribution of $g(i-1)$ in $g(i)$
  Only if $B_{i-1} > B_i$ that each way in $g(i-1)$ contributes one way in $g(i)$

I know the above details are quite hard to understand from the pure text. Let's assume [condition] evaluates to 1 if condition is true, and 0 if false. We can write the above recurrences as follows.

$$f(i) = [B_i \geq B_{i-1}]f(i-1) + [B_i = B_{i-1}]g(i-1)$$
$$g(i) = f(i-1) * w(max(B_{i-1}, B_i + 1)) + [B_{i-1} > B_i]g(i-1)$$

Hence, we can compute the values of $f(i)$ and $g(i)$ one by one in increasing order, and compute the final answer as $f(N-1) * w(B_{N-1}) + g(N-1)$

Questions you may have

- Why choosing these weird functions?
  Functions are a really nice representation of how dynamic programming works. Referring my code, ways[i][0] represents $f(i) and ways[i][1] represents $g(i)$.
- How do I debug this kind of problem?
  The best way would be to write a brute force solution. For this problem, I'd recommend writing brute force for $N = 4$ and $M = 10$, generating random array B and running 4 nested loops to compute the correct answer in $O(M^4)$. I have added such a snippet for your reference.

I hope you guys found it understandable and clear. Refer my code, written on same lines.

PREREQUISITES:

Bitwise Operations, Greedy

PROBLEM:

You are given two arrays A and B, of size N. In one move, you are allowed to swap the values of A_i and B_i for any i. Determine the maximum possible bitwise AND of array A, and the minimum number of moves needed to achieve it.

EXPLANATION:

Observation: The x^{th} bit (in the binary representation) of the final answer is 1 if and only if, the x^{th} bit is 1 in each of the upward facing numbers.
(The proof is a direction application of the definition of bitwise AND)

Define an array S such that, s_i is

- 0, if in the final answer, card i should have number A_i upwards,
- 1, if in the final answer, card i should have number B_i upwards,
- -1, if the side of the card to be facing up is undecided (the default state of the card in this case is number A_i upwards).

initially, all s_i is -1.

Let us tackle the problem one bit at a time. Since we are concerned with maximising the answer, we seek the most significant bit to be as large as possible.
Iterate over the bit range in descending order - from 29 to 0 (largest and smallest possible bits that can be set in the answer, respectively). Let the current bit we are processing be x.

Step 1: Determine if it possible to flip a subset of the *undecided* cards (those with $s_i = -1$) such that every number facing upwards has the x^{th} bit set.

Why? How?

By the definition of s_i, it is clear that, the only cards we can flip are those whose state is undecided. To determine if it is possible to have all upward facing numbers with the x^{th} bit set:

- check if the x^{th} bit is set in the upward facing number of each *decided* card.
- check if the x^{th} bit is set in either A_i or B_i of each *undecided* card.

This can be accomplished easily using bit manipulation (the x^{th} bit is set in number a only if $a \& 2^x \neq 0$).

Step 2: If possible, flip the bare minimum number of *undecided* cards to achieve the same.

How?

By the definition of bitwise AND, it is clear that each upward facing number must have the x^{th} bit set. Thus, iterate over each of the N cards,

- skipping the cards whose state is already *decided*.
- skipping *undecided* cards with both A_i and B_i having the x^{th} bit set.
- setting the final state of the remaining cards such that, in the upward facing number, the x^{th} bit is set. Also, update the values of s_i appropriately.

At the end, the maximum possible answer is the bitwise AND of all upward facing numbers (use the number A_i if s_i equals -1) and the minimum number of cards to flip is equal to the number of cards with $s_i = 1$.

TIME COMPLEXITY:

For each valid bit, we iterate over the N cards twice (once per step). We also iterate over the N cards one last time to determine the answer. The total time complexity is then:

$$O(d * (N + N) + N) \approx O(d * N)$$

where d, the bit range, is 30.

PREREQUISITES:

Dynamic Programming, Strings

PROBLEM:

Gotham City is the home of Batman, and Batman likes to keep his people together.

There are N houses in the city. The resident of the i^{th} house is of type A_i. It is known that people of the same type are friends and people of different types are not friends.

To maintain peace, Batman can do the following type of operation:

- Choose indices i and j $(1 \leq i \leq j \leq N)$ and reverse the substring $A[i,j]$.

Determine the length of the **longest** range of friends after applying **at most** k operations on the string for all $0 \leq k \leq N$.

EXPLANATION:

First let's see how we can increase the length of substrings having same character by the reversal operation as given in the question.
For a string say, s and for character say, x, we first take the longest and 2nd longest substrings having all characters as x. We are taking the longest and 2nd longest since we want to maximize the length of the substring formed in the given operation. Say they are at position i_1 and i_2 with length l_1 and l_2. Assuming i_1 is less than i_2, we can just reverse the substring $s[i_1....i_2 - 1]$ to get substring of length $(l_1 + l_2)$ having all x. We can keep repeating this operation till we get the longest substring that contains all the x that are present in string s. Also we would observe that the number of operations required to get the above longest substring would be the number of substrings of x in the original string which would be always less than n, i.e the length of the string s.

Now coming back to this question, what we can do is repeat the operations as defined in the above paragraph for each character from a to z and for each operation take the length of the maximum possible substring that can be formed among a to z as our answer for that operation.

TIME COMPLEXITY:

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

PREREQUISITES:

Sorting, **Set**, **Pair**

PROBLEM:

Chef's college is conducting online exams, where his camera will be monitored by one or more invigilators during the exam. Chef failed to prepare for the exam on time and decided to google the answers during the exam. The exam starts at time 0 and ends at time F. Chef needs a total of K minutes of googling during the exam in order to pass it. Invigilators monitor his camera at N times of the form $[S_i,E_i]$ (where $0 \leq Si \leq Ei \leq F$), during which he can't google. Chef was resourceful enough to get hold of these times and now he wants to know if he'll be able to pass the exam if he googles the answers during the times when no one is looking at his camera.

QUICK EXPLANATION:

We need to find sum of minutes within time intervals wherein no invigilator is looking at Chef's camera, i.e. count all minutes that do not lie in any of the given N intervals.

Following the order of appearance of the intervals (i.e. sorted order), if the immediate next interval starts strictly after all of the previous intervals have ended, then the time between these 2 points can be used for googling.

If the sum of such times is greater than or equal to the amount of time Chef needs for googling during the exam then he shall pass otherwise he wouldn't have enough time and would fail.

EXPLANATION:

We have to find the number of minutes wherein no invigilation takes place, which can be obtained with the help of intervals where no invigilation is being done (As between times x and y i.e. interval $[x, y]$, there lie a total of $y - x$ minutes). The un-invigilated intervals are of the form $[x, y]$, where the immediate previous interval of invigilation ended at x minutes and the immediate next one starts at y.

Any two invigilation intervals, $[a, b]$ ($b \geq a$) and $[x, y]$ ($y \geq x$) can be of the following forms (considering $x \geq a$, i.e. second interval starts after or at the same time as the first one):

- CASE 1: $x \leq b$ and $y \leq b$, whereby the latter interval lies completely within the former.
  - between the occurrence of the 2 intervals there is no time where invigilation isn't happening. The intervals can collectively be written as $[a, b]$.
- CASE 2: $x \leq b$ and $y > b$, whereby some of the latter interval's initial minutes are also some of the former one's final minutes.
  - no un-invigilated period between the two intervals. They can be collectively written as $[a, y]$.
- CASE 3: $x > b$, whereby the latter interval occurs entirely after the former.
  - the time interval between the two that is not invigilated is $[b, x]$, i.e. $x - b$ minutes of time for Chef to google.

to identify which of the above cases they belong to, we need to sort the intervals by their starting points (in increasing order) and then while traversing them, keep track of the maximum time an interval has ended at so far. The third case only occurs when the immediate next interval to the current one (in the sorted list of intervals) has starting time greater than the ending time of any other previously occurred interval. We maintain a variable to hold the total minutes Chef has had for googling so far and whenever the third case is encountered we add minutes to this variable.

To establish this we use a set of pairs (vector can also be used provided that it is sorted after pushing all pairs), thus storing the given intervals by their starting points (conflict in position of equal starting points in the sorted order is resolved by sorting as per the ending point of the intervals, but it is of no consequence for our approach, all we need is that the starting points be accessed in their order of appearance). Once all the pairs are inserted into the set, we traverse it for finding intervals satisfying case 3.

We keep track of the maximum end point among the intervals that we come across while traversing the set, let's call it x. If the starting point of an interval is found to be less than or equal to x, then no contribution will be made to the minutes Chef has for googling. Whereas when the starting point of an interval (let a be the starting point) is greater than x, $a - x$ will be added to Chef's time.

If we call the starting time of the very first invigilation to be s, then apart from the time **between** intervals that are found while traversing, $F - x$ (time between the end of last invigilation to the end of exam) and $s - 0$ (time between start of exam and start of first invigilation) also contribute to Chef's time.

If this total time calculated is greater than or equal to K, we output *Yes* otherwise Chef would fall short of time to google thus making the answer *No*.

Algorithm using vector of pairs named s to store pairs of starting and ending points of given intervals

```
maxend=0;
for every pair in s:
    if maxend<pair.first:
        ans+=pair.first-maxend;
    if pair.second>maxend:
        maxend=pair.second;

ans+=F-maxend
if ans>=K:
    output "YES";
else:
    output "NO";
```

TIME COMPLEXITY:

$O(N \times \log N)$ per test case.
As complexity of **insertion** in STL set is $O(\log N)$, and insertion is done for N pairs denoting intervals of invigilation.

PREREQUISITES:

Coordinate Geometry

PROBLEM:

While playing in the plains, Chef found a point A with coordinates (X, Y).

Chef is interested in finding the number of **straight lines** passing through the point A such that their **intercepts** on both axes are **positive integers**.

EXPLANATION:

We can see that the required equation of line can be written as

$$\frac{x}{a} + \frac{y}{b} = 1$$

This can be further written as:

$$(a - x)(b - y) = xy$$

Taking $P = (a - x)$, $Q = (b - x)$ and $N = xy$, we get

$$PQ = N$$

Now we just need to find different values of P and Q such that it satisfies the above equality, which can also be viewed as different number of ways as N can be written as a product of two numbers.
To solve this we would do prime factorisation of N. This can be done using sieve or just iterating till square root of x and y.
Finally N can be written as

$$N = p_1^{r_1} \times p_2^{r_2} \times .....p_k^{r_k}$$

where $p_1, p_2.....p_k$ are prime factors of N. Now we would form our number P from $p_1, p_2.....p_k$ and Q would simply be $\frac{N}{P}$, Now to form P, for each i, we can either take p_i^0 or p_i or p_i^2 or…$p_i^{r_i}$. Thus for each i we would have $r_i + 1$ options to choose from. Thus our final answer would be the same as number of ways to form P which can be written as:

$$answer = \prod_{i=1}^{i=k} (r_i + 1)$$

TIME COMPLEXITY:

$O(\sqrt{x} + \sqrt{y})$

PROBLEM:

Tonmoy has a special torch. The torch has 4 **levels** numbered 1 to 4 and 2 **states** (On and Off). Levels $1, 2,$ and 3 correspond to the On state while level 4 corresponds to the Off state.

The **levels** of the torch can be changed as:

- Level 1 changes to level 2.
- Level 2 changes to level 3.
- Level 3 changes to level 4.
- Level 4 changes to level 1.

Given the initial **state** as K and the number of changes made in the levels as N, find the final **state** of the torch. If the final state cannot be determined, print Ambiguous instead.

EXPLANATION:

Lets take this case by case for K:

- K = 0: This corresponds to state 4. Here if the number of changes, i.e N is a multiple of 4 then that means that it would again reach the state of 4 and so the final state would be *off* else if it is not a multiple of 4, then the final state would be among $1, 2, 3$ which would imply the *on* state.
- K = 1: This corresponds to any one of the state among $1, 2, 3$. Here again if the number of changes, i.e N is a multiple of 4, it would mean that the final state would be the same as initial state. Thus it would remain *on* as it was initially. Else if N is not a multiple of 4, then we cannot comment on the final state and so it would be $Ambiguous$.

TIME COMPLEXITY:

$O(1)$ for each test case.

PREREQUISITES:

Dynamic Programming

PROBLEM:

You are given N compounds numbered from 1 to N. The initial amount of the i^{th} compound is Q_i.

You are also given M thermal decomposition equations. Every equation is of the form:
$C_0 \rightarrow W_1 \cdot C_1 + W_2 \cdot C_2 + \ldots + W_X \cdot C_X$.

After **Prolonged heating** find the amount of each compound.

EXPLANATION:

It is simple forward dynamic programming.
Since, $C_{i,j-1} < C_{i,j}$, we can say that a compound will decompose in higher indexed compound.
So, we can safely first decompose compound in order $A_1, A_2, ..., A_N$.

We can complete equations in the order they are given as $C_{i,0} < C_{i+1,0}$.

Let $dp[i]$ be amount of A_i at some point of time.

PseudoCode for dp can be viewed as:
Go from $i = 1$ to M:
  Go from $j = 1$ to X_i:
    Update $dp[C_{i,j}]$ as $(dp[C_{i,j}] + dp[C_{i,0}])$ **mod** $(10^9 + 7)$

Finally, update all $dp[C_{i,0}] = 0$

TIME COMPLEXITY:

$O(M + \sum_{1 \le i \le M}(X_i))$

PREREQUISITES:

Arrays, Optimization

PROBLEM:

You are given a doubly ended queue Q of size M containing elements in the range $[1,N]$ **atleast once**. Find the minimum total number of elements to pop (given we can pop from either side) to get all the N distinct elements in the values we have popped.

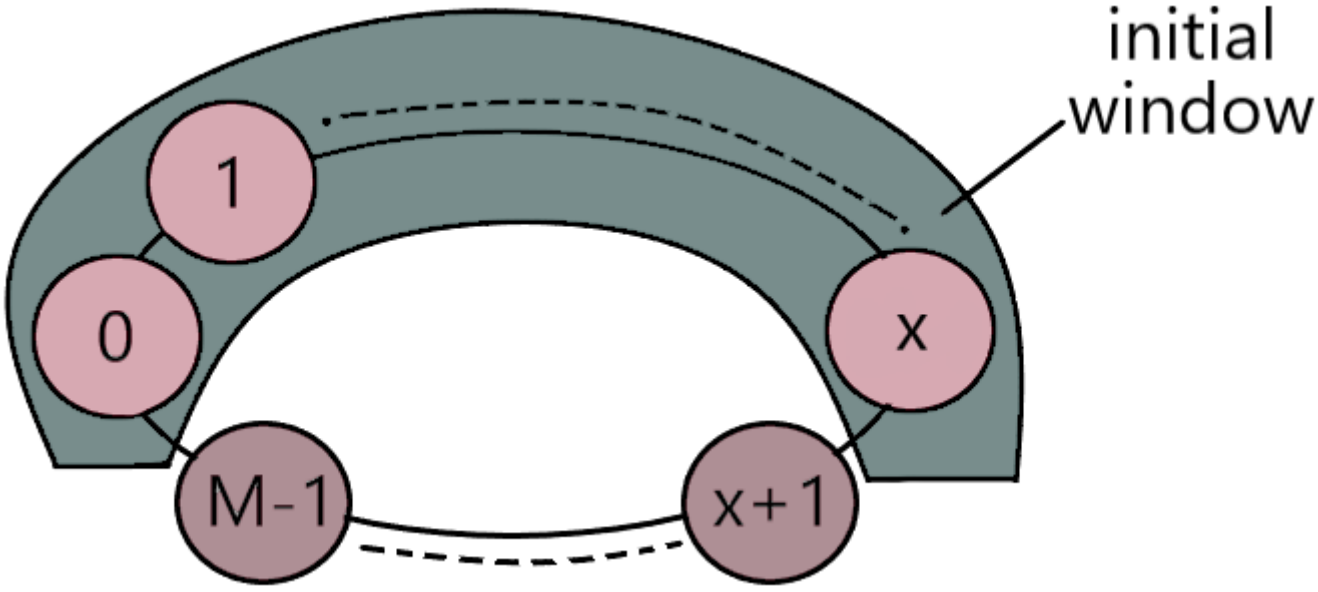
QUICK EXPLANATION:

Visualizing the given dequeue as a circular array starting at index 0 and ending at $M-1$ with 0 and $M-1$ being next to each other, we will find the smallest sub-array (that must include at least one of the two extreme indices of the array) consisting of all N numbers.

EXPLANATION:

Note: The popping of some elements from the beginning and some from the end in a doubly ended queue can be visualized as the removal of a sub array consisting of one or both of the extreme indices of the array from a circular array. This would help in understanding the aforementioned approach. Thus we will be depicting the given array as circular array for ease in understanding.

We first find the smallest sub array starting at index 0, that contains all numbers from 1 to N and record its size as the answer.



Considering 1 to N all numbers are present from indices 0 to X at least once, the initially chosen window would look as above.

We would then remove an element from the rear end of the window, effectively reducing its size by 1 in doing so.

- If this removal causes a deficiency of any number from 1 to N (say x) in this new window:
  We fulfill this deficit by stretching the window from it's front end (which expands backwards i.e. initially starting from index 0, we stretch it towards $M-1$, $M-2$ and so on - similar to expanding a window in a circular array) to find and accommodate x, thus changing the beginning of the window to index $M-i$ (i being the index at which we found x for the first time while moving in the direction of the expansion of the window), and increasing its size. We then record the answer as the minimum of previous answer obtained and the current window size (window being of the form $[\ldots, M-2, M-1, 0, 1, \ldots]$).

- If the removal of last element from the window has not compromised the appearance of any number from 1 to N:
  We record the answer as the size of this new window (as it is guaranteed to be one element smaller than the previous window).

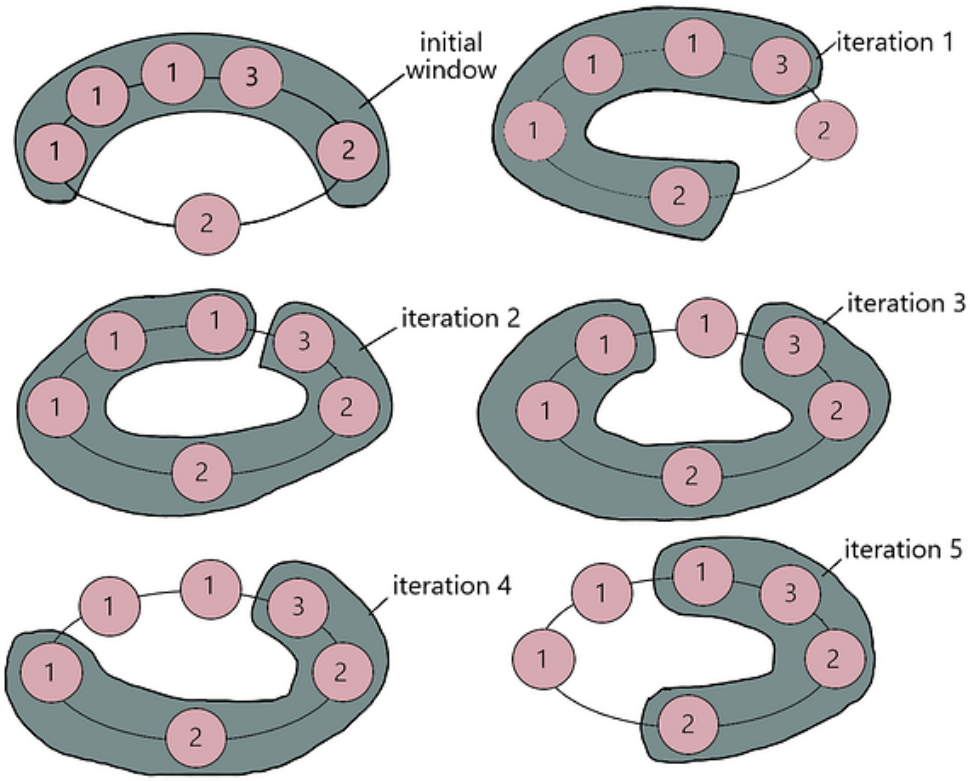
We continue this process of removal and extending window till the index $M-1$ becomes the rear end of the window (window with rear end $M-1$ is the last window for which we perform the iteration).

Why stop at this point?
Algorithm for mentioned approach

Let us visualize the algorithm with the help of one of the test cases given in the question:

```
N = 3, M = 6
Q = {1, 1, 1, 3, 2, 2}
```

The windows found in each of the iterations of the while loop in the algorithm would be as shown in the figure below:



The minimum of all window lengths containing 1 to N numbers at least once is evidently 4, thus the answer is 4.

TIME COMPLEXITY:

$O(M)$ per test case.

PROBLEM:

Given two integeres x and y, we need to find if it is possible to reach (x, y) from $(0, 0)$ if we can perform the following operations any number of times:

- Move coordinate (i,j) to $(i+1,j+1)$
- Move coordinate (i,j) to $(i+1,j-1)$
- Move coordinate (i,j) to $(i-1,j+1)$
- Move coordinate (i,j) to $(i-1,j-1)$

QUICK EXPLANATION:

- If the absolute value of $x+y$ is even, we output YES, else we output NO.

EXPLANATION:

Without loss of generality, we can change (x, y) to $(abs(x), abs(y))$ where $abs(i)$ denotes the absolute value of i.

First, let us observe the following:

Suppose we are at (i,j) and perform the first operation and move to $(i+1,j-1)$. We can observe that in this case the **parity** of $i+j$ doesn't change, for $(i+1,j-1)$ the parity is same as (i,j) since $(i+1+j-1) \mod 2 = (i+j) \mod 2$.

Similarly, we can prove that the parity of $i+j$ doesn't change for other operations also.

We know that for the starting point $(i,j) = (0, 0)$, parity of $i+j = 0+0 = 0$.

From this, we can tell that if parity of $x+y$ is 1, the answer is impossible.

Otherwise we can prove that the answer is always possible.

Let us assume $x < y$. Then we can apply the first operation $\frac{(y+x)}{2}$ times, and then second operation $\frac{(y-x)}{2}$ times. The x coordinate will then become $\frac{(y+x)}{2} - \frac{(y-x)}{2} = x$ and the y coordinate will become $\frac{(y+x)}{2} + \frac{(y-x)}{2} = y$.

Similarly, we can prove this for $x \geq y$.

TIME COMPLEXITY:

$O(1)$ for each testcase.

PROBLEM:

Professor is at some direction and Tokyo is at exact opposite direction to professor. To get the same direction as Tokyo, professor can take any substring of S consisting of L and R characters, and perform those operations. Here L means turning left and R means turning right. We need to determine if professor can turn in the same direction as Tokyo or not.

EXPLANATION:

- Since Tokyo is initially at the exact opposite direction of professor, in order to get to the same direction as Tokyo, Professor can perform two operations of the same type.
- That means professor can perform either LL or RR to get in the same direction as Tokyo.
- Therefore in the string S, if there is a substring of length 2 which has equal characters, then the answer is YES.
- Else the string is either $LRLR\ldots$ or $RLRL\ldots$ Here, whatever substring we take, if we follow the steps in that substring one move's effect will cancelled by the next move. (L and R cancel each other). Therefore, in this case, the answer is NO.

TIME COMPLEXITY:

$O(N)$ for each testcase.

PREREQUISITES:

Sorting or maps/dictionaries

PROBLEM:

A non-empty subsequence of an array is said to be *good* if all its elements are distinct. Given an array A, count the number of *good* subsequences it has.

QUICK EXPLANATION:

Let F_x denote the number of times x occurs in the array. The answer is

$$\prod_x (1 + F_x) - 1$$

where the product is taken over all x which appear in the array.

EXPLANATION:

Let F_x denote the number of times x appears in A.

Checking every subsequence of A for whether it's good or not is obviously going to be too slow, since there are 2^N of them in total.

Let's look at it differently. Consider some good subsequence S. It must have distinct elements, so let's look at them one by one.

- Either S contains 1, or it doesn't. If it does contain 1, there are F_1 choices for which index this 1 is chosen to be. So, in total this gives us $F_1 + 1$ possibilities for 1.
- Either S contains 2, or it doesn't. If it does, there are F_2 choices for the index of 2. Once again, this gives us $F_2 + 1$ possibilities.
- Similarly, it either doesn't contain 3; and if it does contain 3, there are F_3 choices for it. Again, $F_3 + 1$ possibilities for 3.

⋮

So in general, considering some integer x, there are $F_x + 1$ possibilities provided by it — either x is not present at all, or there are F_x possible indices from where to choose x.

These choices are independent for each x, which means the total number of possibilities is $(F_1 + 1) \cdot (F_2 + 1) \cdot \ldots \cdot (F_M + 1)$ where M is the maximum possible number in A (in this case, $M = 10^6$).

Note that this product also allows the choice of the empty subset - where we choose not to take any element. To exclude this case, subtract 1 from the above product.

The complexity of our solution (so far) is $O(M)$ per test case. However, there can be up to 10^5 test cases, and $M = 10^6$ so this isn't quite fast enough.

OPTIMIZING FURTHER

The formula given above correctly computes the answer but is too slow. To optimize it, note that we don't need to care about elements that don't appear in the array, because they always contribute 1 to the product.

This allows us to optimize the computation in several ways, two of which are mentioned below.

Method 1: Maps

By far the simplest way to optimize the computation is to use maps (`std::map` or `std::unordered_map` in C++, `TreeMap` or `HashMap` in Java, `dict` in python) to compute the frequencies of the elements, and then just iterate over these maps. This ensures that we only consider elements that occur in the array at least once, of which there can be at most N.

Building the frequency table takes $\mathrm{O}(N \log N)$ (or expected $\mathrm{O}(N)$, if hashmaps are used), and the computation of the product is $\mathrm{O}(N)$ afterward. The sum of N across all tests is bounded so this is fast enough.

Method 2: Sorting

It is also possible to accomplish this without any fancy data structures, with the help of sorting.
Note that all we really want to do is compute the frequencies of elements of A.
Suppose we sort A. Then, all equal elements will be present adjacent to each other.
This allows us to apply a simple two-pointer algorithm (or binary search) to find all segments of equal elements, and once we know those lengths we simply compute the desired product to solve the problem.

Once again, the time complexity of this is $\mathrm{O}(N \log N)$ for the sort and $\mathrm{O}(N)$ afterward, which is more than enough.

TIME COMPLEXITY:

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

PROBLEM:

An array is said to be good if all its elements are distinct, i.e. no two elements of the array are equal to each other.

You are given a positive integer N and an integer K such that $N \leq K \leq \binom{N+1}{2}$.

Construct an array A of length N that satisfies the following conditions:

- A has exactly K good (contiguous) subarrays, and
- Every element of A is an integer from 1 to N (both inclusive).

EXPLANATION:

Observation

For different types of arrays, check the number of good subarrays present.

- **Array where all elements are equal:** Here, only the subarrays of length 1 are good. Thus, the total number of good subarrays is N, where N is the length of the array.
- **Array where all elements are distinct:** If all the elements are distinct, any possible subarray of this array is good. Thus, the total number of good subarrays is $\binom{N+1}{2}$.
- Any other type of array would have M number of good subarrays, where $N < M < \binom{N+1}{2}$.

The number of good subarrays can be changed by introducing/removing duplicates from certain positions.

Note that, not only the number of duplicates, but their position also matters in determining the number of good subarrays.

Solution

All subarrays of length 1 are good. Thus, we need $K' = K - N$ good subarrays of length ≥ 2. From now on, we consider subarrays of length ≥ 2 only.
An array of length X having distinct elements contributes $\binom{X}{2}$ to the answer. Since the total number of good subarrays is K', we build an array of length X with **distinct elements** such that $\binom{X}{2} \leq K < \binom{X+1}{2}$.

We now need $K' - \binom{X}{2}$ good subarrays. This number would be less than X.

Why?

We know that $K < \binom{X+1}{2}$.
$\Rightarrow K - \binom{X}{2} < \binom{X+1}{2} - \binom{X}{2}$
$\Rightarrow K - \binom{X}{2} < X$

For the $(X+1)^{th}$ and subsequent elements, we can place the number at position $X - (K' - \binom{X}{2})$. This would contribute $K' - \binom{X}{2}$ good subarrays, all ending at position $X+1$. Since all the subsequent elements are duplicates of $(X+1)^{th}$ element, they would not contribute anything.

TIME COMPLEXITY:

The time complexity is $O(N)$ per test case.

PREREQUISITES:

Prime factorization

PROBLEM:

Given an integer $N > 1$, the following operations is performed until N becomes 1:

- Choose an integer $x > 1$ that is divisible by N and divide N by x. If x is even, we get A points, else we get B points.

We need to find the maximum number of points achievable.

EXPLANATION:

Let us first do the prime factorization of N. Let the total sum of even powers be $even$ and the total sum of odd powers be odd.

For example, if $N = 180$ then $N = 2^2 3^2 5^1$. Therefore, $even = 2$ and $odd = 2 + 1 = 3$.

Now the problem can be solved in various cases depending on A and B:

Case 1: $A \geq 0$ and $B \geq 0$

- In this case, since both A and B are non-negative, we want to divide N as many times as possible by both even and odd primes.
- Therefore, in this case, the answer will be $even \cdot A + odd \cdot B$.

Case 2: $A \leq 0$ and $B \leq 0$

- In this case, since both A and B are not positive, we don't want to divide N many times as this could reduce the cost. We want to get it done in one go i.e, divide N by N itself.
- Therefore, in this case, the answer will be A if N is even or else it will be B .

Case 3: $A \geq 0$ and $B \leq 0$

- In this case, we wan't to divide N by 2 as many times as possible. We also wan't to avoid dividing N by odd numbers.
- Suppose y is the maximum odd number divisible by N. If N is even, the first step we could do is divide N by $2 \cdot y$ and then keep on dividing N by 2. Or else $y = N$ and we need to divide N by N itself.
- Therefore, in this case, the answer will be $even \cdot A$ if N is even or else it will be B.

Case 4: $A \leq 0$ and $B \geq 0$

- In this case, we want to divide N by odd primes as many times as possible. We also want to avoid dividing N by even numbers.
- Suppose N is odd. Then we can divide N as many times as possible by odd primes. If N is even, we want to remove the even part in one operation itself. Then we can continue removing odd primes one by one.
- Therefore, in this case, the answer will be $odd \cdot B$ if N is odd or else it will be $odd \cdot B + A$.

PREREQUISITES:

Dynamic Programming

PROBLEM:

Given a sequence X of length M, $f(X)$ is defined as $f(X) = \sum_{i=1}^{M-1} |X_{i+1} - X_i|$.

JJ has an array A of length N. He wants to divide the array A into two subsequences P and Q (possibly empty) such that the value of $f(P) + f(Q)$ is as large as possible. (Note that each A_i must belong to either subsequence P or subsequence Q).

Help him find this maximum value of $f(P) + f(Q)$.

EXPLANATION:

We follow 1-based array indexing for this solution.

Let $dp[i][j]$ denote the maximum value of $f(P) + f(Q)$ such that:

- All the elements till index $max(i, j)$ have been divided into subsequence P and Q
- Subsequence P ends at the i^{th} element and subsequence Q ends at the j^{th} element.

Note that, since the elements at indices i and j lie in different subsequences, $i \neq j$.

The first element which has not been added to any of the subsequences yet is the element at position $k = max(i, j) + 1$.

While adding the k^{th} element to subsequence P, two cases are possible:

- $i > 0$: The last element added to subsequence P was the i^{th} element of the array. Thus, on adding the k^{th} element to subsequence P, we add $abs(A[k] - A[i])$ to $f(P)$. Formally, $dp[k][j] = max(dp[k][j], dp[i][j] + abs(A[k] - A[i]))$.
- $i = 0$: This means that no element was added to subsequence P yet. The first element added to subsequence P is the k^{th} element of the array.
  Thus, $dp[k][j] = max(dp[k][j], dp[i][j] + 0)$.

Similarly, if we add the k^{th} element to subsequence Q, two cases are possible:

- $j > 0$: The last element added to subsequence Q was the j^{th} element of the array. Thus, on adding the k^{th} element to subsequence Q, we add $abs(A[k] - A[j])$ to $f(Q)$. Formally, $dp[i][k] = max(dp[i][k], dp[i][j] + abs(A[k] - A[j]))$.
- $j = 0$: This means that no element was added to subsequence Q yet. The first element added to subsequence Q is the k^{th} element of the array.
  Thus, $dp[i][k] = max(dp[i][k], dp[i][j] + 0)$.

The final answer would be the maximum out of all $dp[i][j]$.

TIME COMPLEXITY:

The time complexity is $O(N^2)$ per test case.

PREREQUISITES

Pre-computation, Prime sieve and a bit of number theory.

PROBLEM

Given a number N, you are allowed to make the following operation any number of times.

- Choose a number K such that K has exactly four divisors, divide N by K.

You want to minimize the number of factors of N. $1 \leq N \leq 10^6$

QUICK EXPLANATION

- Since for all K, $N/K < N$, so we can compute the answer for each N from 1 to N, using previous answers.
- For current N, we can make a list of its divisors having exactly four factors. We can also precompute the number of divisors of each number.
- We can check for each candidate the best answer, the one leading to minimum number of divisors.

EXPLANATION

Since there are 10^6 values of N, we can precompute the answer for each N and then print them out.

First of all, some basic number theory class.
Number of factors of a number
If number $N = \prod p_i^{a_i}$ where p_i are distinct primes, then the number of divisors of N is $\prod(1 + a_i)$

Why?

Let's assume N has K distinct prime factors. Consider an int vector b with K entries, where $0 \leq b_i \leq a_i$. We can see that each such vector correspond to exactly one factor of N. i-th value can take $(1 + a_i)$ different values, and values at each position are independent, so the number of possible vectors b is $\prod(1 + a_i)$

Read in detail **here**

Making list of factors of all numbers up to N can be done in two ways.

- Factoring each number individually (slower)
  This way, we'd spend $O(\sqrt{x})$ time factoring x
- Considering multiples of all values from 1 to N.
  This way, we consider each number v one by one, and iterate over its multiples $k * v$. All multiples of v definitely have v has a factor.
  The time complexity of this approach is $N + N/2 + N/3 \ldots N/N = N * (1 + 1/2 + 1/3 \ldots 1/N)$ which is roughly $O(N * log(N))$

Hence, now we have number of factors of a number as well as the list of factors of each number $1 \leq n \leq N$.

Now, an important thing to observe here is how N changes.

Let's consider $N = 216 = 2^3 * 3^3$. The factors of 216 having exactly 4 divisors are 6, 8 and 27. Considering each K one by one, we can replace $N = 216$ with 36, 27 or 8.

Assuming D_n denotes the number of divisors of N, and A_n denote the minimum number of divisors of N after applying any number of operations, then we can say that $A_n = min(D_n, min_{A_{n/k}})$ where k is a factor of N with

exactly four divisors. D_n denotes not performing operation at all, hence the required number of divisors is the number of divisors of N in that case.

Since we have A_m already computed for all $m < n$, and we also have D_n, we can easily compute this.

For $N = 216$, $A_{216} = min(D_{216}, min(A_{36}, A_{27}, A_8))$. All values D_{216}, A_{36}, A_{27} and A_8 are already computed, hence A_{216} is computed.

Do have a look at setter's solution, if looking for neat implementation.

An alternative way of approaching (Not part of above solution)

As it happens, we didn't use the fact of number having exactly four divisors at all. A number have exactly four divisors in following cases.

- The number is product of two primes p and q where $p \neq q$
- The number is of form p^3 for some prime p

So if we write the prime factorization of $N = \prod p_i^{a_i}$, try noticing what impact does dividing by K have on factorization of N.

If $K = p_i * p_j$, it reduces a_i and a_j by one. If $K = p_i^3$, then it reduces a_i by 3.

Hence, considering only list a, the problem is reduced into following.

Given a list of values a, you are allowed to reduce two distinct elements by one, or reduce any element by 3. You can perform any operation any number of times. Determine the minimum value of product $\prod(1 + a_i)$ after applying operations optimally.

Can anyone figure out a decent approach on how to solve this? Do comment. Have fun as always!

TIME COMPLEXITY

The time complexity is $O(MX * log(MX) + T)$ where $MX = 10^6$.
The space complexity is $O(MX)$ or $O(MX * log(MX))$, depending upon implementation.

PREREQUISITES:

Factorization

PROBLEM:

Given array A, compute the number of unordered pairs (i,j) such that $i \cdot j$ divides $A_i \cdot A_j$.

EXPLANATION:

Create an array of pairs C whose elements are (i, A_i) for all $1 \leq i \leq N$. Let $C_{x,1}$ and $C_{x,2}$ represent the first and second elements of the pair C_x, respectively.

Call a pair of pairs (C_i, C_j) *special* if $C_{i,1} \cdot C_{j,1}$ divides $C_{i,2} \cdot C_{j,2}$. Then. we want to find the total number of unordered pairs (C_i, C_j) that are *special*.

Start by dividing $C_{i,1}$ and $C_{i,2}$ by $\gcd(C_{i,1}, C_{i,2})$, for all i. Let the resulting array of pairs be D.

Claim: (D_i, D_j) is special iff (C_i, C_j) is special.

Proof

For clarity, lets define $C_i = (a, b)$ and $C_j = (c, d)$. Also, let $g_1 = \gcd(a, b)$ and $g_2 = \gcd(c, d)$. Finally, define $D_i = (a', b')$ and $D_j = (c', d')$, where $a = g_1 \cdot a', b = g_1 \cdot b', c = g_2 \cdot c'$ and $d = g_2 \cdot d'$.

Then, (C_i, C_j) is *special* implies

$$\begin{gathered} a \cdot c \mid b \cdot d \implies \\ (g_1 \cdot a') \cdot (g_2 \cdot c') \mid (g_1 \cdot b') \cdot (g_2 \cdot d') \implies \\ a' \cdot c' \mid b' \cdot d' \end{gathered}$$

implying pair (D_i, D_j) is also *special*.

So, the problem is equivalently reduced to finding the number of special unordered pairs of D.

Claim: (D_i, D_j) is special iff $D_{i,1} \mid D_{j,2}$ and $D_{j,1} \mid D_{i,2}$.
(The proof is trivial and left to the reader as an exercise; Use the fact that $D_{i,1}$ and $D_{i,2}$ share no common factors.)

Therefore, if (D_i, D_j) is special, then

$$\begin{aligned} D_{i,1} \mid D_{j,2} &\implies D_{i,1} = D'_{j,2} \\ D_{j,1} \mid D_{i,2} &\implies D_{j,1} = D'_{i,2} \end{aligned}$$

where $D'_{i,2}$ and $D'_{j,2}$ are factors of $D_{i,2}$ and $D_{j,2}$ respectively.

Let $dp_i[x][y]$ represent the number of j $(j < i)$ such that $x = D_{j,1}$ and $y \mid D_{j,2}$.

It is not too hard to see (using the above deductions) that the number of j $(j < i)$ such that (D_j, D_i) is *special* is equal to

$$\sum_{f \in F} dp_i[f][D_{i,1}]$$

where F is the set of all factors of $D_{i,2}$.
Summing this value over all i gives us the required answer!

Implementation

Maintain a 2d hash table (nested `unordered_map<>` in C++) to hold the values of dp_i. Let's call it DP. Currently, hash table DP is empty.

Iterate from 1 to N. Let i be the current index we are processing.

Calculate all factors of $D_{i,2}$ quickly, **using sqrt decomposition**. Then, for each factor f, add $DP[f][D_{i,1}]$ to the answer. Finally, update the hash table by adding 1 to $DP[D_{i,1}][f]$ for all factors f.

TIME COMPLEXITY:

Calculating array of pairs D takes $O(N \log \max(i, A_i)) \approx O(N \log N)$ (the log factor is present because we have to compute the gcd).

Updating the DP table after processing each D_i can be done in $O(\sqrt{D_{i,2}}) \approx O(\sqrt{X})$ time, where $X = \max A_i$.
Assuming constant time lookups and updates to table DP, the total time complexity is equal to

$$O(N \log N + N\sqrt{X})$$

per test case.

PROBLEM:

Alice is teaching Bob maths via a game called N-guesser.

Alice has a number N which Bob needs to guess.

Alice gives Bob a number X which represents the sum of divisors of N. Alice further provides Bob with A and B where A/B represents the sum of reciprocals of divisors of N.

Bob either needs to guess number N or tell that no such number exists.

EXPLANATION:

The divisors of a number n are of the form a_1, n/a_1, a_2, n/a_2, a_3, n/a_3 …sqrt(n) (only if it divides n). So, the sum of reciprocal of the divisors of a number is nothing but the sum of divisors of the number divided by the number n itself.
So, we can the get the number n as X*B/A. But, we have to check that this n yields the same sum of the divisors as given input X, which will require O(sqrt(n)) time.

TIME COMPLEXITY:

O(sqrt(n)) for each test case.

PREREQUISITES:

Greedy

PROBLEM:

Given a budget of P rupees, you may buy the following items (any number of times):

- Fuljhari for a rupees,
- Anar for b rupees,
- Chakri for c rupees

To light up a Anar, you require x Fuljihari's. To light up a Chakri, you require y Fuljihari's. Determine the maximum total of Anar and Chakri's that you can light up.

EXPLANATION:

Since we require x Fuljihari's to light up a Anar, the total cost to buy and light up an Anar is equal to $b + x * a$. Similarly, the total cost to buy and light up a Chakri is equal to $c + y * a$.

Since we are only interested in lighting up the maximum total number of Anar's and Chakri's, we use a greedy strategy and only buy the firecracker with a cheaper total cost.
So, we spend $\min(b + x * a, c + y * a)$ per lit firecracker. Since we have a budget of P rupees, the maximum total number of firecrackers we can light up using this strategy is equal to

$$\lfloor P / \min(b + x * a, c + y * a) \rfloor$$

(where $\lfloor x \rfloor$ equals the largest integer less than x).

TIME COMPLEXITY:

$$O(1)$$

per test case.

PREREQUISITES:

Sorting, Binary Search, Priority Queue or Multisets

PROBLEM:

Given a hallway of length N, you have M workers to clean the floor with each worker responsible for $[L_i, R_i]$ segment (segments might overlap). What's the minimum amount of time required to clean the floor. Every unit of length should be cleaned by at least one worker. If the workers can clean 1 unit of length in unit time and can start from any position within their segment and can also choose to move in any direction. Also, the flow of each worker should be continuous, i.e., they can't skip any portion and jump to the next one, though they can change their direction.

QUICK EXPLANATION:

Idea is to binary search on the time. To check if time t would be enough or not, we need to start from location 1 and then keep moving in right direction until we reach N. For this, initially sort the segments in increasing order of L_i and maintain current set of R_i s sorted in increasing order using a priority queue or multiset. So at current location $curLocation$, we check if we have any R_i such that we can move ahead. We move to the new location of $min(R_i, curLocation + t)$. Finally if we are able to reach the location N, we can finish in time t.

EXPLANATION:

As mentioned in quick explanation, to find the minimum time to finish, we will use the idea of binary search. We can binary search on time so now we need to check if a particular time t is enough or not.

Let's sort the segments initially in increasing order of L_i. So that we can start from location 1 and keep moving in right direction. Because moving in left is not helpful for us. All left side segments are already covered.
Lets also maintain current set of segments S in increasing order of R_i. This will help us to move right. This can be done using a priority queue or multiset.

Now lets start with $curLocation = 1$ and iterate over all workers in sorted order. If the current worker's $L_i \leq curLocation$, we will include it in our current set. And then iterate over our current set of segments S. For any segment $[L, R]$ in S, we will update $curLocation = min(R, curLocation + t)$. And we will erase the current segment $[L, R]$ from S. We will keep removing elements from S until we find one segment which updates our current location or S becomes empty. If we don't find any such segment which updates our location, we will break current iteration loop over workers. Because now we won't able to clean whole floor.

This way we will process all workers and finally check if $curLocation = N$, then we can finish in time t and we will try for smaller t, otherwise we will try for larger t.

TIME COMPLEXITY:

$O(n \cdot \log n \cdot \log m)$ per test case

PROBLEM:

Dazzler has an interesting task for you.

You will be given an array A of N positive integers. N is always even.
Exactly $N/2$ elements in the array will be Even and $N/2$ elements will be Odd.

In one operation you should do the following steps:

- Choose two different indices $i(1 \leq i \leq N)$ and $j(1 \leq j \leq N)$
- Make $A[i] = A[i] + 1$
- Make $A[j] = A[j] - 1$

After doing all the necessary number of operations (possibly none) you need to preserve the parity of each element as in the initial array A. For example if an element at the index $i(1 \leq i \leq N)$ is even, at the end $A[i]$ should be even

You need to find whether you can make all the odd elements in the array equal and all the even elements equal separately.

Print **YES** if it is possible to meet the above conditions after doing any number of operations, else print **NO**.

EXPLANATION:

First thing to observe is that, since we can select any i and j and change $A[i] = A[i] + 1$ and $A[j] = A[j] - 1$, so we can essentially change the array to any other array A' as long as it holds the condition $\sum_{i=1}^{i=n} A'[i] = sum$, where $sum = \sum_{i=1}^{i=n} A[i]$
Now we want the final array A' to be such that $N/2$ of its elements are even, say $2X$ and rest $N/2$ elements to be odd, say $(2Y + 1)$. Thus

$$sum = \frac{N}{2} \times (2X + 2Y + 1)$$

If we subtract $\frac{N}{2}$ from sum, we would have:

$$sum - \frac{N}{2} = N \times (X + Y)$$

Thus for such X and Y to exist, $(sum - \frac{N}{2})$ would have to be divisible by N.
Thus if $(sum - \frac{N}{2})$ is divisible by N then it is possible, otherwise its not possible.

TIME COMPLEXITY:

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

PREREQUISITES:

Basic Array Traversal

PROBLEM:

Alice has recently learned in her economics class that the market is said to be in equilibrium when the supply is equal to the demand.

Alice has market data for N time points in the form of two integer arrays S and D. Here, Si denotes the supply at the i_{th} point of time and D_i denotes the demand at the i_{th} point of time, for each $1 \leq i \leq N$.

Help Alice in finding out the number of time points at which the market was in equilibrium.

EXPLANATION:

The market is said to be in equilibrium when the $demand = supply$ at a particular point of time. The problem wants us to calculate the number of instances of time where the values in the supply array equal those in the demand array.

More formally, Chef wants to find the number of times:
$supply[i]$ = $demand[i]$ ∀, i ∈ s.t. $0 \leq$ i $<$ length(supply or demand array)

TIME COMPLEXITY:

To iterate the two arrays together, we require $O(N)$ time, which is in accordance with the array size.

PREREQUISITES:

Modulus Operation

PROBLEM:

Alice has a standard deck of 52 cards. She wants to play a card game with $K-1$ of her friends. This particular game requires each person to have an equal number of cards, so Alice needs to discard a certain number of cards from her deck so that she can equally distribute the remaining cards amongst her and her friends.

Find the **minimum** number of cards Alice has to discard such that she can play the game.

Note: It doesn't matter which person gets which card. The only thing that matters is that each person should have an equal number of cards.

EXPLANATION:

- Number of players playing the card game is K
- Let minimum number of cards Alice has to discard be M. Number of remaining cards = $52-M$.
- Because each player will have an equal number of cards, the remaining number of cards should be divisible by K.

Updated Problem

Find minimum $M \geq 0$ such that $52-M$ is divisible by K.

Because the constraints are small, we can iterate over M (from 0 to 52), and get the answer.

What if constraints were not small?

Instead of 52, let say we had N cards, where $1 \leq N \leq 10^9$.

So now, we want to find the largest M such that $N-M$ is divisible by K.

$N-M=\lambda \cdot K$, for some $\lambda \geq 0$.

$M=N-\lambda \cdot K \implies M=N$.

SOLUTION:

Tester's Solution

Editorialist's Solution

nairobiny Mar '22

Just to note that the CARDGAME.

Quote

lavish_adm Mar '22

Fixed. Thanks 🙂

PREREQUISITES:

Longest Increasing Subsequence
You do not need to know the algorithm of computing LIS, just a familiarity with the definition of LIS is sufficient.

PROBLEM:

For a permutation P of length N, we define $L(P)$ to be the length of the longest increasing subsequence in P. That is, $L(P)$ is the largest integer K such that there exist indices $i_1 < i_2 < \ldots < i_K$ such that $P_{i_1} < P_{i_2} < \ldots < P_{i_K}$.

Define P^R to be the permutation $(P_N, P_{N-1}, \ldots, P_1)$.

You are given a positive integer N. You need to output a permutation P of length N such that $L(P) = L(P^R)$, or say that none exist.

Note: P is said to be a permutation of length N if P is a sequence of length N consisting of N **distinct** integers between 1 and N. For example, $(3, 1, 2)$ is a permutation of length 3, but $(1, 4, 2)$, $(2, 2, 3)$ and $(2, 1)$ are not.

HINTS:

- In constructive problems, it usually helps to try figuring out the answers for lower values of N using pen and paper. It usually leads to a good number of observations, and helps in constructing the answer for the larger N.
- In the constructive problems involving LIS, thinking in the following two directions can be sometimes useful: Constructing sequence which is first increasing and then decreasing (or vice-versa). Or breaking down the sequence in blocks, and choosing and arranging elements in block such that you can only take a single (or some fixed number of) element/s from each block.

Coming back to the problem, the following hints might be useful:

Hint 1

The answer is “NO” only for $N = 2$. For all other values of N, a valid permutation exists.

Hint 2

Try to find out the answer for $N = 4, 5$ on pen and paper.

Hint 3

For $N = 4$, the valid permutations are: $\{2, 4, 1, 3\}$ and $\{2, 1, 4, 3\}$
For $N = 5$, one of the valid permutations are: $\{1, 2, 5, 4, 3\}$

Hint 4

The answer for odd values of N is: First have an increasing sequence from 1 till $\lfloor \frac{N}{2} \rfloor$, and then have a decreasing sequence from N till $\lceil \frac{N}{2} \rceil$.

Hint 5

Try constructing answer for even values of N using the construction for odd values of N. There can be several possible solutions. One such solution is:

First have an increasing sequence from 2 till $\frac{N}{2}$, then have 1 and then have a decreasing sequence from N till $\frac{N}{2} + 1$.

So, for $N = 6$, the answer will be $2, 3, 1, 6, 5, 4$.

EXPLANATION:

After trying to find the permutation for $N = 5$ while thinking in the general directions as mentioned in the Hints section, we can figure out that the answer for odd values of N (present in Hint 4).

There is actually no algorithm of finding out the construction, you just need to play a little bit with some random permutations, and keep using the observations made in the process.

When N is even

At first, it looks like the answer will be “NO” for even values of N. However, on trying all the possible permutations for $N = 4$, you will see that the following two permutations are valid: $\{2, 4, 1, 3\}$ and $\{2, 1, 4, 3\}$

You can try to expand the solution for $N = 4$ to higher values, and observe the pattern.

An Incorrect way?

After seeing the permutation $\{2, 1, 4, 3\}$, and going by the general direction of breaking sequence in blocks, it initially looks motivating to break the sequence in blocks of 2 and try to proceed further. On further pursuing, you might see some patterns, but they seemed little complicated. Do let us all know in the comment box if you have got some pattern 🙂

One possible way

Because we have a solution for odd values, it looks motivating to get use the answer for $N - 1$ and extend it to N.

One way to do this is to observe that we create Increasing Subsequences at the beginning of the permutation. If we insert 1 in the middle, it will not affect the LIS.

The same argument holds when we look at the reverse of the permutation. This motivates us to create answer for $[2 \cdots N]$ using Hint 4, and then insert 1 in the middle. This leads to the following construction:

First have an increasing sequence from 2 till $\frac{N}{2}$, then have 1 and then have a decreasing sequence from N till $\frac{N}{2} + 1$.

TIME COMPLEXITY:

For each Test-Case, it will take $O(N)$ time.

PREREQUISITES:

MEX

PROBLEM:

You are given an array A containing $2 \cdot N$ integers. Is it possible to reorder the elements of the array in such a way so that the MEX of the first N elements is equal to the MEX of last N elements?

QUICK EXPLANATION:

- Let i be the smallest number having less than 2 occurrences in A.
- The answer is `YES` if i has 0 occurrences.
- The answer is `NO` if i has 1 occurrence.

EXPLANATION:

Let us denote the subarray with first N elements as P and that with last N elements as Q.

We know that, for a particular number i to exist as the MEX of P as well as Q, all j $(0 \leq j < i)$ must have atleast one occurence in P as well as atleast one occurence in Q. This means that there must be atleast two occurrences of j in the array A.

Let i be the **smallest** number such that, the number of occurence of i in A is less than 2.

- **There are 0 occurrences of i in A:** This means that the MEX of both P and Q is i.
- **There is 1 occurrence of i in A:** We can place this i either in P or in Q. If we place it in P, the MEX of Q becomes i. However, the MEX of P would atleast be $(i + 1)$. This means that the MEX of P and Q can never be equal.

Implementation: We store the count of each number in array A. Starting from 0, find the smallest number i having count less than 2. All numbers less than i can be distributed among P and Q such that there is atleast one occurrence in both. If i has count 0, the answer is `YES`. If it has count 1, the answer is `NO`.

TIME COMPLEXITY:

The time complexity is $O(N)$ per test case.

PREREQUISITES:

Bitwise XOR

PROBLEM:

Given a positive integer N, construct two arrays A and B, each of size N such that:

- $A_i \neq A_j$ for all $(1 \leq i < j \leq N)$.
- $B_i \neq B_j$ for all $(1 \leq i < j \leq N)$.
- $A_i \neq B_j$ for all $(1 \leq i, j \leq N)$.
- $1 \leq A_i, B_i \leq 3 \cdot 10^5$ for all $(1 \leq i \leq N)$;
- $\bigoplus A[1, i] = \bigoplus B[1, i]$ for all $(1 \leq i \leq N)$ such that i is **even**.
  Here $\bigoplus A[L, R]$ corresponds to the **bitwise XOR** of all elements of the subarray $[L, R]$ of array A.
  Similarly, $\bigoplus B[L, R]$ corresponds to the bitwise XOR of all elements of the subarray $[L, R]$ of array B.

EXPLANATION:

We define our two arrays A and B as follows:

$$A[i] = 2 \times i + 1, 1 \leq i \leq N$$
$$B[i] = 2 \times i, 1 \leq i \leq N$$

Now if we check XOR of first i elements of A and B where i is even, then we would notice that the first bit in the final xor of both $A[1..i]$ and $B[1..i]$ would be 0. This is because none of the elements of B has their first bit set since they are all even and for A since there are even number of set bits in the first bit position so the first bit would be unset in the final xor. Thus the first bit would be same in the final xor of both A and B. Now if we ignore the first bit in A and B, then essentially

$$A[j] = B[j], 1 \leq j \leq N$$

so their corresponding XOR $\bigoplus A[1...j]$ and $\bigoplus B[1...j]$ would also be the same(ignoring the first bit) , where $1 \leq j \leq N$ and since we already saw above that the first bit for $\bigoplus A[1...i]$ and $\bigoplus B[1...i]$ is also same, therefore

$$\bigoplus A[1...i] = \bigoplus B[1...i]$$

TIME COMPLEXITY:

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

PREREQUISITES:

Maths, Combinatorics

PROBLEM:

You are given an array A consisting of N integers and Q queries. Each query is described by two integers L and R. For each query, output the number of tuples (i, j, k) such that $L \leq i < j < k \leq R$ and $A_i + A_j + A_k$ is an even number.

EXPLANATION:

Since we are looking for even tuples, let us see when will the combination of three numbers result in an even sum. We can easily see that the conditions to get an even sum are:

$$Even + Even + Even = Even$$
$$Odd + Odd + Even = Even$$

Let's say E and O are the count of even and odd numbers respectively within the queried range of positions, $[L, R]$ in the given array.

Calculating E and O in a queried window of A

Running a loop from positions L to R for every query would result in a complexity of $O(N \times Q)$ for calculation of even and odd numbers in the queried window of the array.

Instead, we precompute the number of even numbers from position 1 to x for every $x \in [1, N]$ and store them in an array (say B), i.e. B_x = number of even numbers in $A_{[1,x]}$. This makes it clear that the number of even numbers, E in $A_{[L,R]}$ will be given by B_R when $L = 1$ and by $B_R - B_{L-1}$ otherwise. Similarly the number of odd numbers, O in $A_{[L,R]}$ will be given by $(R - L + 1) - E$.

As in the first type of tuples, we need to choose 3 even numbers from the total even numbers present in the queried window of the given array, the total number of tuples of the first type are:

$$fst = \binom{E}{3}$$

Now in the second type of tuples, we need to choose 2 odd numbers from the total odd numbers along with 1 even number among all even numbers present in $[L, R]$. Hence the total number of tuples of the second type will be:

$$snd = \binom{O}{2} \times \binom{E}{1}$$

Thus the total number of tuples with even sum in the array window $[L, R]$ would become:

$$total = fst + snd$$

That is our final answer for every query, we can simply output it.

TIME COMPLEXITY:

$O(N + Q)$ per test case

PREREQUISITES:

Bitwise Xor

PROBLEM:

A number X is called *bad* if its binary representation contains odd number of 1 bits. For example, $X = 13 = (1101)_2$ is bad while $X = 3 = (11)_2$ is not bad.

Chef calls an array A of length N **special** if the following conditions hold:

- For each $1 \leq i \leq N$, $0 \leq A_i < 2^{20}$
- All the elements of A are **distinct**
- There does not exist any non-empty **subset** of A such that the **bitwise XOR** of the subset is bad.

For example,

- $A = [2, 3, 4]$ is not **special** because the XOR of the subset $[2, 3]$ is $2 \oplus 3 = 1$, which is *bad*. ($\oplus$ denotes the bitwise XOR operation)
- $A = [3, 3]$ is not special because its elements are not distinct.
- $A = [3, 5]$ is special because it satisfies all the conditions.

Chef challenges you to construct **any** special array of length N. Can you complete Chef's challenge?

Hints:

A number X is called *bad* if its binary representation contains odd number of 1 bits.
Let the number X be called *good* if its binary representation contains even number of 1 bits.

Hint 1

When we take XOR of two bits, the total number of 1 bits either remains same, or decreases by 2.

Hint 2

When we take XOR of two numbers, the total number of 1 bits changes by even number.

Hint 3

When we take XOR of set numbers, the total number of 1 bits changes by even number.

Hint 4

Consider any set of *good* numbers. The XOR of the subset will also be a *good* number.

EXPLANATION:

Let us use the defined terminology of *good* and *bad* elements.

Our aim is to construct a **special** array of length N. In other words, we want to create an array of length N containing **distinct** elements, such that there doesn't exist any non-empty subset such that the bitwise XOR of the subset is bad.

Approach 1

Let us first analyze, what happens when we take XOR of two numbers.

Taking XOR of two 1-bit numbers

We have 4 possible cases:

- $0 \oplus 0 = 0$
- $0 \oplus 1 = 1$
- $1 \oplus 0 = 1$
- $1 \oplus 1 = 0$

An important point to note is, the total number of 1-bits remain same in the first 3 cases, and decreases by 2 in the last case.

Taking XOR of two numbers

We have seen that when we take XOR of two bits, the total number of 1-bits either remains same, or decreases by 2. In other words, the total number of 1-bits changes by even number.

When we take XOR of two K-bit numbers, we can consider the operation as taking XOR of two 1-bit numbers K times. And therefore, we can extend the above argument and say that the total number of 1-bits changes by even number.

Taking XOR of set of numbers

Let the set $S = \{S_1, S_2, \cdots, S_K\}$.
The XOR of the set S is defined as $(((S_1 \oplus S_2) \oplus S_3) \oplus \cdots \oplus S_K)$

In other words, it is $K - 1$ successive XORs. We can again extend the same argument as above, and say that when we take XOR of set of numbers, the total number of 1-bits changes by even number.

After making the above observations, let us shift our focus to the subsets.

Each individual element of the array is a valid subset. Let us focus on the subset $\{A_i\}$.
The XOR of this subset is A_i, and therefore A_i should have even number of 1-bits.
This holds for all $i : 1 \leq i \leq N$.

Once we have the above property in the array, by using the observation 3 (Taking XOR of set of numbers), we can claim that XOR of any subset is *good* because the total number of 1-bits in the subset is even (sum of even numbers is even), and after taking XOR, the number of 1-bits changes by even number. Hence, the resulting XOR has even number of set bits.

Finally, we need a list of 1000 *good* numbers. To get the list of *good* numbers, we can start iterating from 0, and check if the number is *good* or not by counting number of set bits.

Can you prove that there are 2^{10} *good* numbers which are less than 2^{11}?

Approach 2

As we have seen in the first approach, each individual element of the array is a valid subset. Let us focus on the subset $\{A_i\}$. The XOR of this subset is A_i, and therefore A_i should have even number of 1-bits.
This holds for all $i : 1 \leq i \leq N$.

We also have to follow the constraint that $A_i < 2^{20}$. In other words, A_i can be a 20-bit number.
Let us divide these 20-bits in two halves, first 10-bits, and last 10-bits.
Now, consider the numbers where first half is exactly same as the second half. Let's call such numbers as *special numbers*. So for example, `1000010000-1000010000` is a *special number*.

Total count of *special numbers*

Let us fill the first 10 bits. Total number of ways to fill them $= 2^{10} = 1024$.
Now, we can replicate these first 10 bits in the last 10 bits.
Hence, total count of *special numbers* $= 1024$

Parity of number of set-bits in a *special number*

Because the first 10-bits are exactly replicated as the last 10-bits, we can claim that the number of set-bits will always be even.

XOR of two *special numbers*

Consider two *special numbers* A and B. Let $A \bigoplus B = C$.
Analyze first 10-bits and last 10-bits of C. They are exactly same!! (Why?)
And therefore, C is a *special number*, and has even number of set bits.

Note that any *special number* can be simply written as $(i \cdot 2^{10} + i)$ where $0 \leq i < 2^{10}$.

The above three statements tells us that if each A_i is a **distinct** *special number*, all the three conditions are satisfied, and we have solved our problem!!

TIME COMPLEXITY:

We can first create the list of 1000 good numbers by iterating from 0 till 2^{11}.
Then for each test case, we can print first N numbers from the list. Hence time complexity is $O(N)$ per testcase.

PREREQUISITES

Contribution Trick , Modular arithmetic

PROBLEM

Let us represent a permutation of numbers from 1 to N by A. Let A_i represents the i^{th} number of the permutation.

We define a function over the permutation as follows:
$F(A) = (A_1 * A_2) + (A_2 * A_3) + \cdots + (A_{N-2} * A_{N-1}) + (A_{N-1} * A_N)$

What is the expected value of the function over all the possible permutations A of numbers from 1 to N?

The expected value of the function can be represented as a fraction of the form $\frac{P}{Q}$. You are required to print $P \cdot Q^{-1} \pmod{1\,000\,000\,007}$.

QUICK EXPLANATION

- For each pair (x, y), the number of permutations where y appear right after x consecutively is $(N-1)!$.
  Hence, it contributes $(N-1)! * \sum_{x=1}^{N} \sum_{y=1, y \neq x}^{N} x * y$
- The above expression can be simplified to $(N-1)! * [(N*(N+1)/2)^2 - N*(N+1)*(2*N+1)/6]$.
  Since we need to compute the expected value over all permutations, we need to divide it by $N!$.

EXPLANATION

Math section

In this problem, we would focus on an individual pair (x, y) appearing consecutively, and try to compute the contribution of pair (x, y) by computing the number of permutations where y appears right after x. Note that pair (x, y) is different from pair (y, x).

Since y must appear right after x, we can consider (x, y) to be a single element. The number of elements becomes $N-1$, so the number of permutations where y appears right after x is $(N-1)!$.

So the pair (x, y) contributes $x * y * (N-1)!$ to answer. Pair (x, x) should not be considered, since x cannot appear twice.

Hence, the sum of contribution of all valid pairs can be written as expression
$S = (N-1)! * \sum_{x=1}^{N} \sum_{y=1, y \neq x}^{N} x * y$

The $x \neq y$ condition is tricky, so let's subtract cases with (x, x).

$$S = (N-1)! * [\sum_{x=1}^{N} \sum_{y=1}^{N} x * y - \sum_{x=1}^{N} x^2] = (N-1)! * [(\sum_{x=1}^{N} x) * (\sum_{y=1}^{N} y) - (\sum_{x=1}^{N} x^2)].$$

The sum of first N integers is $\frac{N*(N+1)}{2}$ and sum of squares of N integers is $\frac{N*(N+1)*(2*N+1)}{6}$

Hence, $S = (N-1)! * [\frac{N*(N+1)}{2} * \frac{N*(N+1)}{2} - \frac{N*(N+1)*(2*N+1)}{6}]$.

This is the sum of values of each consecutive pair over all permutations. But we need to compute the expected value over all $N!$ permutations. We just need to divide S by $N!$.

Hence, the required answer is $\frac{1}{N} * [\frac{N*(N+1)}{2} * \frac{N*(N+1)}{2} - \frac{N*(N+1)*(2*N+1)}{6}]$

Implementation

We need to know about Modular arithmetic in order to actually compute this mod 10^9+7. This **blog** explains the idea, and **this** shares some code.

You also need to compute modular multiplicative inverse to divide expression by N, which can be computed by **fermat's little theorem**.

TIME COMPLEXITY

The time complexity is $O(log_2(N))$ per test case.

PREREQUISITES:

Dynamic Programming, Binary Exponentiation, Fermats Little Theorem

PROBLEM:

Define the sequence f_i as:

- $f_0 =$ "0"
- $f_1 =$ "1"
- $f_i = f_{i-1} + f_{i-2}$ for $i \geq 2$, where $+$ is the string concatenation operation.

Determine the sum of the digits of all subsequences of f_i modulo $10^9 + 7$.

EXPLANATION:

It is evident that f_i only consists of zeros and ones. Let k_i denote the number of ones in f_i.
Also, let len_i denote the length of string f_i.

Claim: Every digit in f_i occurs in exactly $2^{len_i - 1}$ subsequences of f_i.
(See this for details. Complete proof is trivial and left to the reader.)

The sum of the digits over all subsequences is equivalent to the sum of the number of times each 1 occurs in a subsequence. From the above claim, it is clear that the answer is then $k_i * 2^{len_i - 1}$.

All that remains is to calculate k_i and len_i efficiently.

Since f_i is the concatenation of f_{i-1} and f_{i-1}, k_i is therefore equal to $k_{i-1} + k_{i-2}$. Similarly, len_i is equal to $len_{i-1} + len_{i-2}$. This can be precomputed quickly using dynamic programming.

Recurrence relations

Special emphasis must be made on the method to compute the answer modulo $MOD = 10^9 + 7$.
The important point to note here is that, while computing the answer modulo MOD,

$$k_i * 2^{len_i - 1} \equiv (k_i \% MOD) * (2^{(len_i - 1) \% (MOD - 1)} \% MOD) \mod MOD$$

(**not** $2^{(len_i - 1) \% MOD}$) by application of Fermats little theorem (see this for details).
You should use binary exponentiation to compute the value of 2^{len_i} in $O(\log len_i)$.

TIME COMPLEXITY:

Precomputing the answer for every $i \leq 10^5$ can be done in

$$O(n * log(MOD))$$

where $MOD = 10^9 + 7$. The $\log$ factor is present because of the `pow` function (which implements binary exponentiation for an optimal runtime).

PROBLEM:

Chef has a work plan for the next N hours, and for each hour he knows if he must work or can take a break. He can take a nap if there are at least K consecutive hours of break time. What is the maximum number of naps he can take?

EXPLANATION:

Suppose there are x hours of consecutive break time. Chef needs at least K hours to take a nap, so in x hours, he can take at most $\lfloor \frac{x}{K} \rfloor$ naps, where $\lfloor a \rfloor$ denotes the floor of a, i.e, the largest integer less than or equal to a.

So, the maximum number of naps Chef can take can be computed as follows:

- Let the (maximal) segments of break time in Chef's schedule have lengths $x_1, x_2, \ldots, x_m$ (a segment is said to be maximal if it is not contained in another segment of break time which has strictly larger length).
- The maximum number of naps is then $\lfloor \frac{x_1}{K} \rfloor + \lfloor \frac{x_2}{K} \rfloor + \ldots + \lfloor \frac{x_m}{K} \rfloor$

For example, if Chef's schedule is `'0010110001110000'` and $K = 2$, the maximal segments of free time have lengths $2, 1, 3, 4$ from left to right. The maximum number of naps is then $\lfloor \frac{2}{2} \rfloor + \lfloor \frac{1}{2} \rfloor + \lfloor \frac{3}{2} \rfloor + \lfloor \frac{4}{2} \rfloor = 1 + 0 + 1 + 2 = 4$.

In most languages, computing $\lfloor \frac{x}{K} \rfloor$ can be done by **integer division**, that is, just `x / K`. Note that python users need to use `x // K` instead.

Finding the maximal segments of break time can be done in $O(|S|)$ by iterating over the string and keeping a variable denoting the length of the current segment, as in the following pseudocode:

```
cur = 0
for i from 1 to N
    if S[i] = '0'
        cur += 1
    else
        // cur is now the length of a maximal segment, do whatever you want with it
        // ...

        // reset the length to 0
        cur = 0
end
// at the end of the loop, cur holds the length of the maximal segment of break time which
```

TIME COMPLEXITY:

$O(N)$ per test case.

PROBLEM

Given a $N \times N$ grid, fill the grid with -1 and 1 such that following conditions are met:

- For **every** column - (Sum of values present in the column) $\times$ (Product of values present in the column) < 0
- For **every** row - (Sum of values present in the row) $\times$ (Product of values present in the row) < 0

QUICK EXPLANATION

- For even N, filling the whole grid with -1 satisfies all the conditions.
- For odd N, filling the main diagonal with 1 and the rest of the grid with -1 satisfies the conditions.

EXPLANATION

Since this is a constructive problem, there may exist multiple solutions. Feel free to share yours in the comments.

Since we want the sum of elements $\times$ product of values to be negative for all rows and columns, one tempting idea would be to fill the whole matrix with -1. Let's see what happens in this case.

- For even N, the product of each row and column is 1, and the sum of each row and column is $-N$, so we get the sum $\times$ product to be $-N$ for each row and column. This satisfies all the conditions.
- For odd N, the product of each row and column is -1, and the sum of each row and column is $-N$, so we get the sum $\times$ product to be N for each row and column. This doesn't satisfy all the conditions.

So, for odd N, we have to use 1 somewhere. Changing the value of one cell changes the sum of the row to be $-(N-2)$ and product of row to 1, hence sum $\times$ product becomes $-(N-2)$. Since $N \geq 2$, smallest odd N is $N = 3$, hence $-N+2$ is always a negative value for given constraints.

Hence, for each row, it is sufficient to replace one value with 1. We need to do that for columns as well. We also need to make sure that no two chosen cells share a row or column, otherwise, the product of that row or column would become -1 again.

One neat way of doing this is to flip the value from -1 to 1 for all cells in the main diagonal. All rows and columns get covered, with no row or column having more than one flipped cell.

For example, for $N = 5$, the grid looks like

```
 1 -1 -1 -1 -1
-1  1 -1 -1 -1
-1 -1  1 -1 -1
-1 -1 -1  1 -1
-1 -1 -1 -1  1
```

The Sum of each row and column is $-N+2$ and the product of each row and column is 1.

TIME COMPLEXITY

The time complexity is $O(N^2)$ per test case since we need to print output of size N^2.

PROBLEM:

Given a positive integer N and an integer k where $0 \leq k \leq N$, we need to find a permutation p of numbers from 1 to N which has exactly k fixed points. A fixed point is defined as an index i for which $p_i = i$.

QUICK EXPLANATION:

- If $k = N$, we output the array $1, 2, \ldots N$.
- If $k = N - 1$, we output -1, since the N^{th} point will automatically be fixed once we fix $N - 1$ points.
- If $k < N - 1$, we output the array as follows: $1, 2, 3, \ldots k, k + 2, k + 3 \ldots N, k + 1$.

EXPLANATION:

There are many ways to go about it. One such way is as follows:

- If $k = N$, then simple we need to output the array $1, 2, 3, \ldots N$.
- If $k = N - 1$, that means there are $N - 1$ fixed points, then automatically the remaining point will be fixed having N fixed points. Therefore, this case is impossible and we ouput -1.

Otherwise we are left with $k < N - 1$, for which we can prove that answer is always possible by constructing the array as follows:

First let us fix the values $p_i = i$ for $1 \leq i \leq k$. Now we have our k fixed points and the remaining points (from $k + 1$ to N) must not be fixed.

To achieve this, what we can do is to simply perform left cyclic shift of $k + 1, k + 2, \ldots N$.

After performing this left cyclic shift we get $k + 2, k + 3, \ldots N, k + 1$ and store these in indices $k + 1, k + 2, \ldots, N$ respectively. Clearly, $p_i = i + 1$ for $k + 1 \leq i < N$ and $p_N = k + 1 < N$.

Therefore, we have our answer which is $1, 2, 3, \ldots, k, k + 2, k + 3, \ldots, N, k + 1$.

TIME COMPLEXITY:

$O(N)$ for each testcase.

PREREQUISITES:

Binary Search, Greedy, Observations

PROBLEM:

A bird starts at a height H at $x = 0$, and there are N obstacles i^{th} of which is of height h_i at position x_i. The only difference is that the obstacles are only from bottom to top and not in the reverse direction. For the bird to cross the i^{th} obstacle successfully, the bird's height at x_i should be strictly greater than h_i.

Find minimum number of clicks required to cross all the obstacles given that with one click if initially at (i, j), then after the click $(i + 1, j + 1)$, otherwise if we don't click at $x = i$, then at $(i + 1, j - 1)$. Also at any point before $x = x_N$, height of the bird should remain non negative otherwise it will drown.

QUICK EXPLANATION:

Notice that if we keep ascending the bird continuously for first k moves and let it descend such that it crosses all the obstacles, thats equivalent to simultaneous ascend + descend in an optimal game play. So the above problem can be solved using binary search on k.

Or we can also solve greedily. Initially decrease H from each h_i. Now just take $max((h_i + x_i + 2)/2)$ for all i. If at any point $x_i <= h_i$ then its impossible.

EXPLANATION:

Binary Search Solution

Let's say in the optimal play, the bird is able to pass through all obstacles with k clicks then clicking k times in the start is always optimal.

Why doing all clicks in the start is optimal?

If we click k times then height would be $H + k$ after all the clicks and after another k steps, it would be again H, because each move H decreases by 1 if we don't click.
In any other strategy where we don't click once, the bird would never have more height than H after $k * 2$ moves. Hence best optimal way is to do all clicks continuously in the start.

Now how to check if k clicks are enough to pass all the obstacles?
We can simulate the process by doing all clicks in first k moves and checking if at every obstacle i, the bird height is > h_i. Now how to calculate height at position x?
If $x_i <= k$, then height would be $H + x_i$ because we have clicked x_i clicks so far.
If $x_i > k$ then height would be $H + k - (x_i - k)$ because we have clicked k times and moved $x_i - k$ extra steps after all the clicks.

Pseudocode

```
def check(k):
    for i in range(n):
        if x[i] <= k:
            if h[i] >= H + x[i]:
                return False
        else:
            if h[i] >= H + k - (x[i] - k):
                return False
```

```
    return True
```

Now if we know that k clicks are enough, then more than k clicks would also lead to a win. Hence we should try for less than k clicks. This gives a perfect situation for binary search on k.

Greedy Solution

The bird is initially at $(0, H)$. Let's bring it to $(0, 0)$ by shifting the x-axis H units up. After this initial position of the bird would be $(0, 0)$ and all the obstacles height would decrease by H.

Now when is it impossible to win?
When $h_i >= x_i$ for any i then its impossible to win because we can click at most x_i times and the bird can achieve at most height x_i but still it would be less or equal to the height of the obstacle hence the bird will crash.

Now how to find minimum number of clicks required?
If we know how many clicks each obstacle would take so that the bird doesn't clash with it, we can take maximum of all to find the number of clicks to safely pass all obstacles.

Now how to find the number of clicks required for i^{th} obstacle which is at position x_i and height h_i?

- We know that the bird started at $(0, 0)$ so to reach x_i, it would need at least $x_i/2$ clicks. One click can make the bird move for two steps. But if we do exactly $x_i/2$ clicks, it would touch $(x_i, 0)$.
- Now we also want the bird to be at height at least h_i so that it doesn't touch the obstacle.

Reaching at x_i and having the height h_i is equivalent to moving $x_i + h_i$ distance. But we need to ensure we don't touch this point, hence it would at least $(x_i + h_i + 1)/2$ clicks if $x_i + h_i$ is odd and $(x_i + h_i)/2 + 1$ clicks if $x_i + h_i$ is even. Or we can write it as $(x_i + h_i + 2)/2$ clicks in both cases.

So finally iterate over the all obstacles and take maximum of $(x_i + h_i + 2)/2$ over all i.

TIME COMPLEXITY:

$O(nlog(10^9))$ for binary search solution.
$O(n)$ for linear solution.

ROBLEM:

Given a binary string S. You may perform the following operation any number of times - choose a previously unselected number X, and flip some substring of S with length X.

Determine a sequence of moves that sorts S in non-decreasing order.

EXPLANATION:

Hint 1

The binary string comprising of all 0's (or all 1's) is in non-decreasing order.
Can you apply the operations to convert the string into either of the above?

Hint 2

Try iteratively converting an increasing prefix of S into all 0's or all 1's.

Solution

We solve with the help of an example.

Consider $S = 1110110010$. A valid sequence of moves is given below:

```
1110110010 // Original string
0000110010 // Invert prefix of len 3
1111110010 // Invert prefix of len 4
0000000010 // Invert prefix of len 6
1111111110 // Invert prefix of len 8
0000000000 // Invert prefix of len 9
```

More formally, we repeatedly invert the longest prefix of S consisting of the same value, until the string is all 0's or all 1's. It is easy to see that the lengths of the inverted substrings are increasing, and hence distinct.

This can be implemented efficiently by iterating over S (from left to right) and appending range $[1..i]$ to the set of inverted substrings when $S[i] \neq S[i+1]$.

TIME COMPLEXITY:

Since we iterate over the string only once, the total time complexity is

$$O(N)$$

per test case.

PREREQUISITES:

Greedy

PROBLEM:

You are given a binary string S_i of length N, where each $S_i = 1$ denotes a person at position i.
Two people are considered friends if the absolute difference of their positions is $\leq K$. A group of people are considered a friend group if each person in the group has at least one friend in the same group.

You can move each person atmost one step from their current position (multiple people can be at the same position after the movements). Determine the minimum number of friend groups you can have if you move the people optimally.

EXPLANATION:

(See the bolded texts in each case for a quick editorial 👅)

We solve this problem greedily.

Iterate over characters of string S from left to right. When the current element is 0, continue iterating. Let x be the current index. Also, let $last$ be the position of the last encountered person while iterating. Initially, let $last = -1$. Finally, maintain another variable ans, which is initially zero.

Case 1: If $last = -1$, it means person at position x is the first person from the left. Thus, since there are no people to his left, it is better to **move this person right** and bring him closer to people on the right.

Case to show why this is necessary

Then, update the value of $last$ to $x + 1$ (since the last encountered person is now at index $x + 1$). Also, set ans to 1 (Signifies the start of a friend group).

Case 2: If $x - last < K$, moving the current person to the right by one step won't break friendship with the last encountered person (on his left), and following similarly from the logic of Case 1, it is wise to **move this person to the right.**

Then, update the value of $last$ to $x + 1$ (since the current person is now at index $x + 1$). Leave ans as it is (since there is no change to the number of friend groups).

Case 3: If $x - last = K$, it is best to **leave the current person where he is**. This is because moving him to the left does no benefit (already part of the last encountered person's friend group) and moving him to the right does no benefit either (he *may* become friends with the next person in line, at the cost of losing his current friend group; so there is no overall decrease in the number of groups).

Then, update the value of $last$ to x (since the current person is now at index x). Leave ans as it is (since there is no change to the number of friend groups).

Case 4: If $x - last = K + 1$, the optimal strategy would be to **move the current person to the left**. Best case, he can maintain friendship with people both to his immediate left and right. Worst case, moving to the left would make it impossible for him to be friends with the person to his right.
Contrasted with the best and worse cases of not moving/moving to the right (where you can at best, only have friendship with the person on your right), it is best to move to the left.

Then, update the value of $last$ to $x - 1$ (since the current person is now at index $x - 1$). Leave ans as it is (since there is no change to the number of friend groups).

Case 5: If $x - last > K + 1$, it is best to **move to the right**, following the logic of Case 1 (not possible to be friends with anyone on the left).

Then, update the value of $last$ to $x + 1$ (since the current person is now at index $x + 1$). Add 1 to ans, since there is now a new friend group.

TIME COMPLEXITY:

Since there is only one iteration done over the string S, the time complexity per test case is

$$O(N)$$

PROBLEM

Sasuke and Itachi are playing a game. Sasuke first creates an array A containing N positive integers $A_1, A_2, \ldots, A_N$. He then creates a new array B of length N such that $B_i = \gcd(A_1, A_2, ..., A_i)$ for each $1 \leq i \leq N$. Now, Sasuke gives array B to Itachi and asks him to find any array A (with $1 \leq A_i \leq 10^9$) such that the given process applied to A will produce B. Can you help Itachi solve this problem?

Here, $\gcd$ stands for **greatest common divisor**.

QUICK EXPLANATION

If the given array B is a valid candidate for A, then print B. Otherwise, no such A exists.

EXPLANATION

Checking if some A exists

We know that $B_i = gcd(A_1, A_2, ..., A_i)$ and $B_{i+1} = gcd(A_1, A_2, ..., A_i, A_{i+1}) = gcd(gcd(A_1, A_2, ..., A_i), A_{i+1}) = gcd(B_i, A_{i+1})$

This way, we can write $B_{i+1} = gcd(B_i, A_{i+1})$.

What just happened?

$gcd(x, y, z) = gcd(gcd(x, y), z) = gcd(x, gcd(y, z))$ holds for all valid x, y, z. So we decided to compute gcd of first i terms, and then take gcd with A_{i+1}. Since gcd of first i terms is B_i, we can write $B_{i+1} = gcd(B_i, A_{i+1})$

The implication of this is that B_{i+1} must be a factor of B_i, Since gcd of two numbers is divisor of both numbers. Hence, condition B_{i+1} divides B_i should hold for all $1 \leq i < N$.

So if the given array has any such i where B_{i+1} doesn't divide B_i, no such A can exist.

Finding A

The given array B is a valid candidate for A. We'd end up with something like $B_{i+1} = gcd(B_i, A_{i+1})$, but we have $A_{i+1} = B_{i+1}$. Since B_{i+1} divide B_i, $gcd(B_i, B_{i+1}) = B_{i+1}$

So the given array B satisfies our condition and can be printed as array A.

TIME COMPLEXITY

The time complexity is $O(N)$ per test case.

PREREQUISITES:

bitwise XOR operation

PROBLEM:

A permutation P of length N (1-based) is called good if it satisfies the following these conditions:

- For each $1 \leq i \leq N$, $P_i \neq i$
- $|P_1 - 1| \oplus |P_2 - 2| \oplus \ldots \oplus |P_N - N| = 0$

Here, $\oplus$ denotes the bitwise XOR operation and $|X|$ denotes the absolute value of X.

Find any good permutation P of length N or print -1 if no such permutation exists.

Note: A permutation of length N is an array which contains every integer from 1 to N (inclusive) exactly once.

EXPLANATION:

The problem can be divided into two cases:

Case 1: N is even. Start with the identity permutation $i.e$ $1, 2, 3....N$ and now start pairing the numbers from the beginning and swapping them $i.e$ 1 is swapped with 2, 3 is swapped with 4 and so on. The bitwise XOR of two same values is 0 and for each index i of the permutation the value of $|P_i - i| = 1$. Therefore the XOR value calculated by $|P_i - i|$ for each pair of adjacent values is 0.This means the total XOR is 0. Swapping ensured that for each $1 \leq i \leq N$, $P_i \neq i$. Therefore this approach satisfies both the conditions.
Case 1: N is odd. If $N = 1$ or $N = 3$ the answer is -1. This can be easily checked by brute force calculation. For $N = 5$ there exists a permutation which satisfies both these conditions which can be found easily by iterating over all permutations of length 5. For easier implementation one can use *next_permutation*. Now when $N \geq 5$ we split the problem into two parts that is for the first 5 elements and rest $N - 5$ elements. As N is odd, $N - 5$ is even. So, we will use the approach mentioned in the case when N is even for the part containing $N - 5$ elements starting from the 6 element $i.e$ 6 is swapped with 7, 8 is swapped with 9 and so on. For first 5 elements we have already calculated the answer.
Output them together in order $i.e$. answer calculated for first 5 elements followed by answer for the rest $N - 5$ elements to get the final answer.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Sorting

PROBLEM:

Chef wants to impress Chefina by giving her the maximum number of gifts possible.

Chef is in a gift shop having N items where the cost of the i^{th} item is equal to A_i.
Chef has K amount of money and a 50% off discount coupon that he can use for **at most one** of the items he buys.

If the cost of an item is equal to X, then, after applying the coupon on that item, Chef only has to pay $\lceil \frac{X}{2} \rceil$ (rounded up to the nearest integer) amount for that item.

Help Chef find the **maximum number** of items he can buy with K amount of money and a 50% discount coupon given that he can use the coupon on **at most one** item.

EXPLANATION:

Let M be the maximum number of items that Chef will buy, and let $\{i_1, i_2, \ldots i_M\}$ be the list of items that costs minimum among all possible list of items of size M. Also, let $S = \sum_{\alpha=1}^{M} A_{i_\alpha}$

Observation 1: Chef will always use the discount coupon on the most expensive item. This is because the amount of money that Chef will pay will be $S - discount$, and as the price increases, the discount increases.

Observation 2: The list of items will have the M lowest priced items. To prove this, let us introduce a new item j in the list of items and remove i_α such that $A_{i_\alpha} < A_j$. We can make 4 cases on {discount on A_{i_α}, discount on A_j} and in each case, we can see that the original list results in less overall cost.

So we can first sort the items in the increasing order of price, and for each i such that $1 \leq i \leq N$, check if we can take first i elements by applying discount on the i^{th} item. The largest feasible i will be our answer.

TIME COMPLEXITY:

$O(N \cdot \log N)$ for each test case.

PREREQUISITES:

Map data structure

PROBLEM:

You are given 2 arrays A and B of length N each. Determine the number of good pairs.

A pair (i,j) is said to be good if:

- $i < j$
- $A_i = B_j$
- $A_j = B_i$

EXPLANATION:

Iterate over the arrays A and B from $j = 1$ to N and keep count of the pair formed by (A_j, B_j). Let us now count the number of indices i for each index j $(i < j)$ such that (i,j) form a good pair.
If (i,j) form a good pair then :
$A_j = B_i$ and $B_j = A_i$;
This means the pair (A_i, B_i) is same as the pair (B_j, A_j);
Thus for each j add count of the pair (B_j, A_j) to the answer and increase the count of (A_j, B_j) by 1

TIME COMPLEXITY:

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

PREREQUISITES:

Dynamic Programming

PROBLEM:

Given a grid of size $N * N$, some of whose cells are *impassable*. You are in the top left cell and want to reach the bottom right cell, moving only one step to the right or downwards, in each move. Each cell also has a certain number of coins you collect on passing through it.

You may choose a sequence of at most K adjacent cells and make them passable. Determine if it is possible to reach the bottom right cell, and the maximum amount of coins you can collect.

EXPLANATION:

Note: We use 0 based row and column indexing in the editorial.

When $K = 0$, the problem is a slight modification of a **classical grid DP problem**. We build our solution directly atop it (readers are requested to get acquainted with the DP solution of the above problem).

Intuition for the DP states

Let $dp[i][j][k]$ be the maximum number of coins we can collect if we move till cell (i, j) and have already made k **adjacent** cells on our path *passable*. Initially, all values are $-INF$.

The only base case - $dp[0][0][0] = A_{i,j}$ is obvious.
The transition states of the DP are (out-of-bounds states are $-INF$):

- $dp[i][j][k] = \max(dp[i-1][j][k-1], dp[i][j-1][k-1]) + A_{i,j}$

  We use the special ability and make cell (i, j) passable, irrespective of whether it is already passable or not.

- $dp[i][j][0] = \max(dp[i-1][j][0], dp[i][j-1][0]) + A_{i,j}$

  Only when cell (i, j) is passable by default. We walk through the cell normally, without having travelled on any special cells before.

- $dp[i][j][K+1] = \max(dp[i-1][j][K], dp[i-1][j][K+1], dp[i][j-1][K], dp[i][j-1][K+1]) + A_{i,j}$

  Only when cell (i, j) is passable by default. We walk through the cell normally, having already used the special ability on cells before this cell.

The final answer is then $\max(dp[N-1][N-1][0], dp[N-1][N-1][1], \dots)$ or -1 if the answer is negative (implying no valid path exists).

TIME COMPLEXITY:

Since computing each state of the DP takes $O(1)$, and computing the final answer takes $O(K)$, the total time to compute the DP table is:

$$O(N * N * K + K) \approx O(N^2 K)$$

PREREQUISITES:

Arithemetic Progression

PROBLEM:

Chef's professor is planning to give his class a group assignment. There are $2N$ students in the class, with distinct roll numbers ranging from 1 to $2N$. Chef's roll number is X.

The professor decided to create N groups of 2 students each. The groups were created as follows: the first group consists of roll numbers 1 and $2N$, the second group of roll numbers 2 and $2N-1$, and so on, with the final group consisting of roll numbers N and $N+1$.

Chef wonders who his partner will be. Can you help Chef by telling him the roll number of his partner?

EXPLANATION:

Note that the roll numbers form an **Arithemetic Progression** with a common difference of 1. Consider the following **Arithemetic Progression** of roll numbers: $1, 2, 3, 4, ..X...2N$.
Now, the teacher pairs students from the start and end. By the properties of AP, all the pairs will have sum = $2N+1$.
Let Chef's partner have a roll number y. Thus, the pair will look like (X, y).
$\implies X + y = 2N + 1$
$\implies y = 2N + 1 - X$

Hence, chef's partner has a roll number y, i.e $2N + 1 - X$.

TIME COMPLEXITY:

The above calculation can be done in constant time. Hence, the solution has a time complexity of $O(1)$.

PREREQUISITES:

Recursion, Binary Search

PROBLEM:

Chef creates a set from a permutation $(P_1, P_2, \ldots, P_N)$ of length N, as follows:

- Initially Chef takes a empty set $S = \phi$, and then for every index i $(1 \leq i \leq N)$, Chef insert a pair of integers (l_i, r_i) into S, where $1 \leq l_i \leq r_i \leq N$ and $(P_{l_i}, P_{l_i+1}, \ldots, P_{r_i})$ denotes the **longest subsegment** of the permutation with P_i as the maximum element.

You are given a set T containing N distinct pairs of integers. Find the number of different permutations of length N from which Chef can create a set S, which is equivalent to set T. Since the number can be very large, print it modulo $(10^9 + 7)$.
Note that the elements of the set T may not be given in a particular order.

QUICK EXPLANATION:

- Start with finding the position of the maximum element. The position of the maximum element is i, iff $[1, i-1]$ and $[i+1, N]$ also exist in the set.
- Let $f(L, R)$ denote the number of different ways to arrange $(R - L + 1)$ numbers under given conditions. The answer to our problem is $f(1, N)$. Use recursion to find this value.
- Let i be the position of the maximum element in $[L, R]$, then:
  $f(L, R) = ((f(L, i-1) \cdot f(i+1, R))\%mod \cdot (\binom{R-L}{i-L}))\%mod.$
  Here, $f(L, i-1)$ and $f(i+1, R)$ are independent to each other. Also, $\binom{R-L}{i-L}$ denotes the ways in which we can choose the numbers for the first half.

EXPLANATION:

Observation

The most important observation in this problem would be to note that, if a pair $[L, R]$ exists in the set, then, for the subarray $[L, R]$, the position of the maximum element in the subarray is **fixed**.

How?

Let us assume that the maximum element occurs at position i, $(L \leq i \leq R)$. We claim that there exists a pair $[L, i-1]$ $(L < i \leq R)$ and a pair $[i+1, R]$ $(L \leq i < R)$. These pairs correspond to the maximum elements in the subarrays $[L, i-1]$ and $[i+1, R]$ respectively.

Amongst all possible values, the i for which the pairs $[L, i-1]$ and $[i+1, R]$ exist in the set, is the position of the maximum element. Note that there would be only **one** such i since the set of pairs was generated from a permutation of numbers.

Finding the position of the maximum element in a pair

If we check all possible values in a pair $[L, R]$ to look for the position of the maximum element, it would be too slow and result in TLE.

Observe that if the maximum element in $[L, R]$ exists at position i, then, there can be no possible pair with value $[L, j]$ $(i \leq j < R)$. In other words, $(i-1)$ is the maximum value for the right bound among all pairs which start at L and end before R.

Similarly, there can be no possible pair with value $[j, R]$ $(L < j \leq i)$. Thus, $(i + 1)$ is the minimum value for the left bound among all pairs which end at R and start after L.

Based on this observation, we can use binary search to find the value of $(i - 1)$ or $(i + 1)$ and then find i.
Calculating the answer for a given pair

For a given pair $[L, R]$, let i denote the position of the maximum element. We now have two subproblems, finding answers to $[L, i - 1]$ and $[i + 1, R]$. This indicates the use of **recursion**.

Let $f(L, R)$ denote the number of different ways to arrange $R - L + 1$ numbers. Since i is the position of the maximum element, we have two subarrays, $[L, i - 1]$ and $[i + 1, R]$. We choose $(i - L)$ elements from remaining $(R - L)$ elements (after excluding element at i) for the first subarray. The elements for the second subarray are assigned simultaneously. We then calculate the answers to both subproblems using recursion. The answer to $f(L, R)$ is nothing but the number of ways of arranging numbers in the first subarray times that of second subarray, times the number of ways of dividing numbers amongst the two subarrays.

Formally, $f(L, R) = ((f(L, i - 1) \cdot f(i + 1, R))\%mod \cdot \binom{R-L}{i-L})\%mod$
Finally, the answer to the problem is nothing but $f(1, N)$.
See setter's solution for implementation details.

TIME COMPLEXITY:

Note that we don't need to visit any pair more than once. Also, for each pair, we run a binary search to find the maximum element's position. Since there are N pairs and binary search runs in $O(logN)$ complexity, the overall complexity is $O(NlogN)$ per test case.

PROBLEM:

Given N candidates appeared for the test, and each of them faced M problems. Each problem was either unsolved by a candidate (denoted by 'U'), solved partially (denoted by 'P'), or solved completely (denoted by 'F').

To pass the test, each candidate needs to either solve X or more problems completely, or solve $(X-1)$ problems completely, and Y or more problems partially. Check if given candidates passed the test or not.

QUICK EXPLANATION:

Just check the conditions as specified in the problem.

EXPLANATION:

For a candidate, lets denote the count of problems solved completely by C_F, solved partially by C_P. Now check if $(C_F \geq X)$ or $(C_F \geq X-1$ and $C_P \geq Y)$.

TIME COMPLEXITY:

$O(N * M)$ if there are N candidates and M problems as we need to iterate through each candidate and find the count of problems solved full, solved partial.

PREREQUISITES:

Trees

PROBLEM:

Chef is the king of the kingdom named Chefland. Chefland consists of N cities numbered from 1 to N and $(N-1)$ roads in such a way that there exists a path between any pair of cities. In other words, Chefland has a **tree** like structure.

Out of the N cities in Chefland, K $(1 \leq K \leq N)$ are considered *tourist attractions*.

Chef decided to visit exactly $(K-1)$ out of K *tourist attractions*. He wishes to stay in the city i $(1 \leq i \leq N)$ such that the **sum of distances** of the **nearest** $(K-1)$ *tourist attractions* from city i is as **minimum** as possible.

More formally, if Chef stays at city i and visits $(K-1)$ *tourist attractions* denoted by set S, then, the value $\sum_{j \in S} D(i,j)$ should be **minimum**.
Here, $D(i,j)$ denotes the distance between cities i and j which is given by the number of roads between cities i and j.

Find the city in which Chef should stay. If there are multiple such cities, print the one with the **largest** number.

EXPLANATION:

This problem can be divided into two separate problems:

- Calculate, for each node, sum of distances of tourist locations from it.
- Calculate, for each node, maximum distance of a tourist location from it.

Assume S' to be the set of all tourist locations.
Lets talk about the first problem now. Assume a function f_sum , such that:

$$f\_sum[i] = \sum_{j \in S'} D(i,j)$$

In order to calculate it we introduce another function $f\_subtree$ where $f\_subtree[i]$ tells us the number of tourist locations in the subtree of i.

We can calculate $f\_sum[1]$ using DFS. For other nodes we can do another DFS and use the following relation

$$f\_sum[child] = f\_sum[parent] + (k - f\_subtree[child]) - f\_subtree[child]$$

This concludes with our problem one. Now moving on to the 2nd part of the problem.
Lets introduce another function f_max such that

$$f\_max[i][0] = maximum\ distance\ of\ tourist\ location\ within\ subtree\ of\ i.$$

$$f\_max[i][1] = maximum\ distance\ of\ tourist\ location\ outside\ subtree\ of\ i.$$

Now we can run a DFS and for any node and its child, if $f\_subtree[child] > 0$ then

$$f\_max[node][0] = max(f\_max[node][0], f\_max[child][0] + 1)$$

Thus we would get first part of the function. For the 2nd part i.e to calculate $f\_max[node][1]$, we run another DFS, where for any $node$ and its $child$, let us take another set C which denotes all children of $node$ other than $child$, then

$$f\_max[child][1] = max(f\_max[node][1], max(f\_max[c][1], where\ c \in C))$$

Now that we solved both these problems individually, we can get back to our original problem. If we assume the required sum of distance for each node i is $answer[i]$, then

$$answer[i] = f\_sum[i] - max(f\_max[i][0], f\_max[i][1])$$

Now we can loop through to find the maximum i with minimum value of $answer[i]$.

TIME COMPLEXITY:

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

PREREQUISITES:

Dynamic Programming, Queue

PROBLEM

Chef hates homework and wants to spend the minimum amount of time possible doing homework. Luckily, he has a few tricks up his sleeve. Chef can hack the school records on any given day to make it show that the assigned homework is complete. However, to avoid suspicion, Chef will not hack the records strictly for more than k days consecutively.

Can you help Chef find the minimum possible total number of minutes that Chef needs to spend on homework?

EXPLANATION:

It is obvious that whenever the number of consecutive days for hacking into records is not less than the number of days the homework is given for, Chef can hack into the records all the days and he would have to spend no time actually doing any homework. So whenever $N \leq K$, the answer would be 0.

For other cases, Chef has 2 choices for each day i where $i \in [1, N]$

- Either he can do his homework spending time H_i
- Or hack into records spending 0 units of time.

As it is given that Chef can't hack the records for more than K days consecutively, we know that if Chef is doing his homework on day i, then he must have done his homework the last time somewhere between days $[i - K - 1, i - 1]$.

We shall be approaching the problem using dynamic programming, as it displays both properties of DP:

Optimal substructure

Let dp_i ($i \in [1, N]$) denote the minimum time spent by Chef on homework upto day i, if he chooses to do his homework on day i. To minimize the time he will be spending on this homework so far (upto day i inclusive), we will choose the last day he did his homework on (say x) from amongst days $[i - K - 1, i - 1]$, such that dp_x has the minimum value amongst all the days in the mentioned interval.

Thus dp_i gives the optimal solution (minimum time Chef spent on homework), if Chef were to do his homework on day i. The minimum amongst the optimal solutions ranging from day $i - K - 1$ to $i - 1$ were in turn considered for finding the optimal solution upto day i. This successive optimization can be traced back to the very first day.

Since the optimal solution for every day can be found using the optimal solutions to the previous $K + 1$ days, the problem presents optimal substructure.

Overlapping subproblems

When we are at day i, we can complete the homework on one of the days $x \in [i - (K + 1), i - 1]$ possessing minimum value of dp_x, and then do the homework on day i.

Similarly when at day $i + 1$, we can complete the homework on one of the days $x \in [i - K, i]$ possessing minimum value of dp_x, and then do the homework on day $i + 1$. We can observe that the intervals in which we search for the minimum value of dp_x coalesce from $[i - K, i - 1]$. Which means we shall be computing the minimum in a bulk of ranges over and over again, thus giving rise to overlapping subproblems.

So far we know how to complete the array dp (read optimal substructure details for introduction to the array named dp throughout the explanation). dp_i for $i \in [1, N]$ holds minimum time Chef would need to complete his

homework if he does his homework on the i_{th} day. Which gave us the following formula:

$$dp_i = min(dp_{i-k-1}, dp_{i-k}, \ldots, dp_{i-1}) + H_i$$

Now let us discuss the most efficient way to resolve overlapping subproblems:

As we traverse through all H_i for $i \in [1, N]$, we use a queue (say dq) to keep track of the minimum value in dp, in a window of length $K+1$, ending at day number $i-1$ i. e. $min(dp_{i-k-1}, dp_{i-k}, \ldots, dp_{i-1})$ along with the day that each value corresponds to in H. For example, in the solution given in this editorial, a dequeue of pairs has been used to store pairs of $\{prev_i, x\}$, where $min(dp_{i-k-1}, dp_{i-k}, \ldots, dp_{i-1})$ is denoted by $prev_i$ and x is the position (day of appearance) of $prev_i$ in H.

Here's how we obtain $prev_i$ values for successive windows while simultaneously updating dp_i without using extra time:

1. Initializing the array dp and queue dq by solving for first window:

   Let us start with the first window. We assign values to dp_i as we traverse H from $i \in [1, K+1]$ as well as build dq to suffice for finding $prev_i$ throughout the first window of dp simultaneously. The first $K+1$ values of dp are equal to the first $K+1$ values of H, because no time could have been taken to complete the homework before the first day resulting in $prev_i + H_i$ to be equal to $0 + H_i = H_i$. With each assignment of dp_i in this range of days, we pop values from dq that are greater than dp_i and then push dp_i itself into dq.

2. Building solutions for successive prefixes of H from second window onwards:

   From $i = K+1$ onwards, we start assigning values to dp that are determined using dq. In dq, we look for elements that are from before day $i-K-1$ and remove if any are found, as that would have increased the window beyond length $K+1$. After this we extract the frontal element in dq which will give us the value $prev_i$, and assign it to dp_i after adding H_i to it.

   To ensure that performing this sequence of operations on dq gives us the minimum element in the next window of dp too, we now remove all values from dq that are greater than the value of dp_i, as any window after the current one will have $prev_{i+1}$ given by $min(prev_{excl}, dp_i)$, where $prev_{excl}$ denotes $prev_i$ excluding the first element of its window. After this is done we insert dp_i in dq as well and continue this process for consecutive indices after i as ending points of successive windows.

Obtaining final answer:

Now that we have the entire array dp, the first day on which it is possible to have done homework for the last time is the day $N-K$ because after that day, there is K days worth of homework that Chef can alter the records for. Moreover, after this *first day of last homework* has been encountered, all days after it are possible last days for him to having done his homework. This results in the answer being minimum of the last $K+1$ values in dp i.e. optimal answer of the last window of dp. This in turn can be obtained from the value of $prev_{n+1}$, which is calculated by using step 2 as explained above for the last time (only for extraction of the frontal element).

TIME COMPLEXITY:

$O(N)$ per test case

PROBLEM:

Ved started playing a new mobile game called Fighting Clans. An army of N enemies is approaching his base. The i^{th} enemy has H_i health points. An enemy gets killed if his health points become 0.
Ved defends his base using a weapon named `Inferno`. He can set the `Inferno` to one of the following two modes:

- Single-Target Mode: In one second, the `Inferno` can target **exactly one** living enemy and cause damage of at most X health points.
- Multi-Target Mode: In one second, the `Inferno` can target **all** living enemies and cause damage of 1 health point to each of them.

Find the **minimum** time required to kill all the enemies.

Note: Ved is **not allowed** to change the mode of the weapon once it is set initially.

EXPLANATION:

Let us calculate answer for both modes:

- Single Target Mode: Here we would loop through each enemy and calculate the time required to kill him. Let us assume a function $f(i)$ that tells the time taken to kill the i_{th} enemy then,

$$f(i) = \lceil \frac{H_i}{X} \rceil$$

$$single\_mode\_answer = \sum_{i=1}^{i=n} f(i)$$

- Multi Target Mode: Here the answer would be the maximum H_i among all the enemies since each second it is reducing the health of all enemies by 1. Thus,

$$multi\_mode\_answer = max(H_i), 1 \leq i \leq n$$

Once we calculate these two, our final answer would be:

$$answer = min(single\_mode\_answer, multi\_mode\_answer)$$

TIME COMPLEXITY:

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

PREREQUISITES:

Binary search

PROBLEM:

We are given an array A of N integers. We need to find the number of pairs (i,j) where $1 \leq i,j \leq N$ and $L \leq CONC(A_i, A_j) \leq R$. Here $CONC(A_i, A_j)$ is defined as the number obtained by concatination of A_i and A_j.

QUICK EXPLANATION:

- Let $len(x)$ be the number of digits in x.
- Get the mathematical expression $CONC(A_i, A_j)$ in terms of A_i, A_j and $len(A_j)$. We will get $CONC(A_i, A_j) = A_i \cdot 10^{len(A_j)} + A_j$.
- Iterate over j from 1 to N, fix A_j and then compute total number of A_i possible from the derived mathematical expression and the bounds mentioned in the problem statement. Then, we will get that A_i must be present in some range which can be solved by sorting the array and performing binary search.

EXPLANATION:

Let us first sort the array A.

Let $len(x)$ be the number of digits in the number x. For example, $len(12345) = 5$.

Let us try to formulate $CONC(x,y)$ in terms x, y and $len(y)$. $CONC(x,y)$ is nothing but the number xy. This can be written as $x \cdot 10^{len(y)} + y$. For example, if $x = 123$ and $y = 45$,
$xy = 12345 = 123 \cdot 10^2 + 45$.

Now, the problem can be solved simply by fixing A_j. From our conditions we have,

$$CONC(A_i, A_j) \leq R$$

$$\implies A_i \cdot 10^{len(A_j)} + A_j \leq R$$
$$\implies A_i \cdot 10^{len(A_j)} + A_j \leq R$$
$$\implies A_i \cdot 10^{len(A_j)} \leq R - A_j$$
$$\implies A_i \leq \frac{R - A_j}{10^{len(A_j)}}$$

Similarly, from the condition involving L, we get $A_i \geq \frac{L - A_j}{10^{len(A_j)}}$.

Therefore, we can solve the problem by simply iterating over j from 1 to N, find the range where our A_i could be i.e, $\frac{L - A_j}{10^{len(A_j)}} \leq A_i \leq \frac{R - A_j}{10^{len(A_j)}}$. Then, we need to find the total number of elements within this range which could be done simply by performing binary search.

TIME COMPLEXITY:

$O(N \log N)$ for each testcase.

PREREQUISITES:

You should be familiar with the concepts of Binary Search and Graph Traversal algorithms like Breadth First Search and Depth First Search

PROBLEM:

*This problem is similar to the problem **"INTREECTIVE2"**. The only difference between them is the **number of queries** allowed — in this problem, up to **21 queries** can be made. In **Div. 1 and Div. 2** this problem is worth **40 points** and **"INTREECTIVE2"** is worth **60**. In **Div. 3** this problem is **non-scorable**.*

This is an interactive task

You are given a tree consisting of N nodes, numbered from 1 to N. Chef selected an arbitrary node and marked it.

Chef gave you a judge to play with, using which you can ask queries about the given tree.

In each query, you give the judge a non-empty set S of nodes. The judge returns 1 if there exists u and v (**not** necessarily distinct) such that $u, v \in S$ and the marked node lies on the shortest path from u to v, or 0 otherwise.

You have to find the marked node.

For each test case, you can use at most **21 queries**.

QUICK EXPLANATION

Because we have 1000 nodes, we can do one complete Binary Search on the nodes in 10 queries. We have 21 queries, so we can do 2 Binary Searches.

Also, let us consider Node 1 as the root node.

Solution 1

- Let us consider the set of leaf nodes $\{L_1, L_2 \dots L_K\}$. With the first Binary Search, we can find the leaf node L_i such that the node in the shortest path from 1 to L_i.
- After finding this lead node, we can do one more Binary Search on the nodes that lie in the shortest path from 1 to L_i.

Solution 2

- Let's find the height of all the nodes using BFS. Now, first we can do a Binary Search on the height of the Node - say H.
- Once we know the height of the marked node, we can do the Binary Search on the nodes which have their height as H to find out the marked node.

PREREQUISITES:

You should be familiar with the concepts of Binary Search and Graph Traversal algorithms like Breadth First Search and Depth First Search

PROBLEM:

*This problem is similar to the problem **"INTREECTIVE"**. The only difference between them is the **number of queries** allowed — in this problem, up to **11 queries** can be made. In **Div. 1 and Div. 2** this problem is worth **60 points** and **"INTREECTIVE"** is worth **40**. In **Div. 3** this problem is **non-scorable**.*

This is an interactive task

You are given a tree consisting of N nodes, numbered from 1 to N. Chef selected an arbitrary node and marked it.

Chef gave you a judge to play with, using which you can ask queries about the given tree.

In each query, you give the judge a non-empty set S of nodes. The judge returns 1 if there exists u and v (**not** necessarily distinct) such that $u, v \in S$ and the marked node lies on the shortest path from u to v, or 0 otherwise.

You have to find the marked node.

For each test case, you can use at most **11 queries**.

Constraints

- $1 \leq T \leq 1000$
- $1 \leq N \leq 1000$
- The input graph is a tree
- Sum of N does not exceed $5 \cdot 10^4$ over all test cases
- At most 11 queries can be made.

QUICK EXPLANATION

Because we have 1000 nodes, we can do one complete Binary Search on the nodes in 10 queries. We have 11 queries, so we can do only one Binary Search.

Also, let us consider Node 1 as the root node.

Solution 1

- Let's do a DFS on the tree starting at the node 1. As soon as we visit a node for the first time, we insert that node into an array arr.
- Analyze this array arr. If we consider the shortest path from 1 to $arr[i]$, then all the nodes that lie in that path are already present in the array as $arr[j]$, such that $j < i$.
- We can do a Binary Search on this array to find the marked node.

Solution 2

- Let's consider a component of the graph having $N/2$ nodes, such that for any two nodes (u, v) in this component, all the nodes that lie in the shortest path from u to v lies in this component.
- Ask a Query in this component. If answer is Yes, we now have $N/2$ nodes to consider. Otherwise we can replace this component by a dummy node, and consider the new graph of $N/2 + 1$ nodes.

PROBLEM:

You are given an integer K.

Consider an integer sequence $A = [A_1, A_2, \ldots, A_N]$.

Define another sequence S of length N, such that $S_i = A_1 + A_2 + \ldots + A_i$ for each $1 \leq i \leq N$.

A is said to be *interesting* if $A_i + S_i = K$ for every $1 \leq i \leq N$.

Find the **maximum** length of an interesting sequence. If there are no interesting sequences, print 0.

QUICK EXPLANATION:

- Calculate A_1, A_2 and so on in terms of K by substituting $i = 1, 2$ and so on in the equation $A_i + S_i = K$
- A_i will come out to be $K/2^i$.
- So the largest value of N such that A_N is an integer will be equal to the largest power of 2 that divided K.

EXPLANATION:

In the problem statement, it is given that:

1. $S_i = A_1 + A_2 + \cdots + A_i$ for each $1 \leq i \leq N$.
2. $S_i + A_i = K$ for each $1 \leq i \leq N$.

Let us see what happens when $i = 1$:
$S_1 = A_1$
$\implies S_1 + A_1 = 2 \cdot A_1 = K$
$\implies A_1 = K/2$

Now, Let us substitute $i = 2$, and use the value of A_1:
$S_2 = A_1 + A_2$
$\implies S_2 + A_2 = A_1 + 2 \cdot A_2 = K$
$\implies 2 \cdot A_2 = K/2$
$\implies A_2 = K/4$

We can continue the process, and see that $A_i = K/2^i$. This can be rigorously proved using **Strong Induction**.

We have the constraint that A_i is an integer for all $1 \leq i \leq N$. This means that we can create the sequence as long as $K/2^i$ is an integer. In other words, the maximum value of N is equal to the highest power of 2 that divides K.

Let the highest power of 2 that divides K be M.
$2^M \leq K \implies M \leq \log_2 K$

To calculate this highest power, we can continue dividing K by 2 as long as it is divisible.

TIME COMPLEXITY:

To calculate the highest power of 2 that divides K, we will require $O(M)$ time.
Therefore, Time Complexity for each test case will be $O(\log_2 K)$.

PREREQUISITES:

Hashing

PROBLEM:

Each color has an integer ID from 1 to N. There are M lists where each color belongs to exactly one list. Batman can distinguish colors belonging to different lists, but he cannot distinguish colors belonging to the same list.

Given a strip of L colors, find the different number of segments Batman will see as a result of his disability. Two positions of the strip are said to belong to the same segment if they are adjacent on the strip and Batman cannot distinguish their colors.

QUICK EXPLANATION:

- In the given strip, replace the Color ID with the List ID to which the color belongs.
- Remove the adjacent duplicates in the strip.
- Output the count of the elements that are remaining in the strip.

EXPLANATION:

We need to find out the number of segments that Batman would be able to see in the given strip. Let us try to find out the starting point of each of the segments. It is quite clear that the number of starting points will be equal to the number of segments in the strip since each segment will have a unique starting point.

A segment will begin from index i of the strip if the colors present at indices i and $i-1$ belong to different lists. Because if they come from the same list, then they are considered to be part of the same segment and then i can't be the starting point of a new segment.

So we are left with finding out the number of starting points possible. To do so, for every index $i \in [2, L]$ we need to find out the list IDs for the colors present at the index i and at index $i-1$. One way is to check every list and find out which one this color belongs to. But this is slow enough to give us a TLE verdict.

We can improve our solution by using hashing. We can simply hash the Color ID with the List ID it belongs to. And then we can easily find out the List ID in constant time for any color. Then when we find the starting point we can simply increment our answer.

Note that the value of number of starting points (i.e. the variable that holds our answer) is initialized from 1, because the first color on the strip will always be the starting point of the first segment that batman can see.

TIME COMPLEXITY:

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

PROBLEM:

Lav has an array A of size N. He noticed that Chef is initially standing at the **first** index of the array.

While standing at the i^{th} index $(1 \leq i < N)$ of the array, Chef can perform the following types of jumps:

- `Jump 1`: Jump to the immediate next index j such that Ai and Aj have the **same parity**.
- `Jump 2`: Jump to the immediate next index j such that Ai and Aj have **different** parity.

Given that Chef can perform `Jump 2` **at most once**, Lav wants to find the **minimum** number of jumps required by the Chef to reach the **last** index of the array.

EXPLANATION:

For each test case, the input consists of an array of size N.

Two cases are possible for this particular problem:

- When the first and last elements have **same** parity.
- When the first and last elements have **different** parity.

In the first case, if we perform Jump 2 even once then it is impossible for us to reach the destination(last element). So the number of jumps will be simply **total number of elements in the array with same parity as the first element** (obviously excluding the first element).

In the second case, we have to **perform Jump 2 once** which can be done at any element having the same parity as the first element. The total number of jumps in this case will be the jumps required to reach this element plus the number of jumps to reach the last element. *This will be equivalent to the number of elements having the same parity as the first element till a particular element plus then the number of elements with different parity till the last element from here on.* So, we to have to find the **minimum value of this sum** for any element in the array having the same parity as the first element. This can be achieved using two extra arrays and pre-computing these values.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Dynamic Programming

PROBLEM:

in every century, there is a chosen one. Therefore, we do not live in a society
–Last paragraphs of the IZhO 2022 Editorial

K0Kalaru47 was out with his friends and he started to tell them about a CP Task. Because the statement was informal, his friends quickly parried with "Who Asked", so here is the formal statement:

Given arrays P and C of size $(N+K)$ each, the *beauty* of an array B of size $(N+K)$ is defined as the sum of C_i for all $1 \leq i \leq (N+K)$ such that $B_i = P_i$.

More formally, the *beauty* of an array B is equal to $\sum_{i=1}^{|B|} C_i \ [B_i = P_i]$

Given an array A of size N such that $(1 \leq A_i \leq M)$.
Your task is to insert K elements into the array such that:

- Each inserted element lies in the range $[1, M]$.
- The *beauty* of the final array is **maximized**.

Note that, an element can be inserted at any index of the array.

You should find two values:

- The **maximum** beauty that can be obtained across all arrays of size $(N+K)$.
- The **number of distinct arrays** of size $(N+K)$ having the maximum beauty. Note that, two arrays X and Y are said to be different if there exists some index i such that $X_i \neq Y_i$. Since the number of distinct arrays can be huge, print the value modulo MOD.

QUICK EXPLANATION:

We can say some final vector B can be obtained from the initial vector if and only if

- All number in B are $[1, M]$.
- Initial vector A is a subset of final vector B.

We can check if the initial vector is a subset of final vector using the following greedy approach.
FIrst pick a pointer $p = 1$. Loop through the array, if at any i, $B_i = A_p$, increment p by 1. If at the end $p = |A| + 1$, then and only then is A a subset of B. This is also known as the Subset Search Algorithm
One important thing to note here is that the subset it finds is always unique. Using this we can define our $dp[i][j]$ to store the maximum beauty and the number of arrays to have that beauty.

One more thing we can notice is that we will form array B by only using the following values:

- P_i
- A_j
- A_{j+1}
- Any other value other than above three.

Thus we can only try these values and have $O(1)$ transitions. Our final answer would be then $dp[n+k][n]$.

EXPLANATION:

Let us construct our $dp[i][j]$ and define it to store two values as:

- The maximum beauty of an array that has length i and the subset search algorithm found the first j values of vector A.
- The number of ways to have that beauty for an array that has length i and the subset search algorithm found the first j values of vector A.

Now let us define the transitions of our dp as follows:

- if $p[i] = a[j]\ and\ a[j] = a[j+1]$:

$$dp[i][j] = max(dp[i][j], \{dp[i-1][j-1].first + C[i], dp[i-1][j-1].second\}, \{dp[i-1][j].first, dp[i$$

- Else if $a[j] \neq a[j+1] \& a[j] \neq p[i] \& a[j+1] \neq p[i]$:

$$dp[i][j] = max(dp[i][j], \{dp[i-1][j].first + C[i], dp[i-1][j].second\}, \{dp[i-1][j].first, dp[i-1][j].$$

- Else if $a[j] = a[j+1]$:

$$dp[i][j] = max(dp[i][j], \{dp[i-1][j].first + C[i], dp[i-1][j].second\}, \{dp[i-1][j].first, dp[i-1][j].$$

- Else if $a[j] = p[i]$:

$$dp[i][j] = max(dp[i][j], \{dp[i-1][j-1].first + C[i], dp[i-1][j-1].second\}, \{dp[i-1][j].first + C[$$

- Else if $a[j+1] = p[i]$:

$$dp[i][j] = max(dp[i][j], \{dp[i-1][j].first, dp[i-1][j].second * (m-2)\}, dp[i-1][j], dp[i-1][j-1]$$

TIME COMPLEXITY:

$O((N+K) \cdot N)$ for each test case

PROBLEM:

A binary string S of length N is said to be a $K-$ *balanced string* if for every index i where $S_i = 1$, there must exist an index j where $1 \leq j \leq N$ and $| i - j | = K$. We need to find the minimum number of operations to convert S into a $K-$ *balanced string* where in a single operation we can choose any index i and flip the value at i .

QUICK EXPLANATION:

- Split the string S into K strings and store them in array $vals$ where $vals_i = S_i S_{i+k} S_{i+2k} \ldots$ for $0 \leq i < K$.
- Now the final answer will sum over the minimum number of operations for each i to convert $vals_i$ into a $1-$ *balanced string*.
- To convert some string T into a $1-$ *balanced string*, we could iterate over i from 1 to N. If $T_i = 1$ and none of the neighbors of T_i have value 1, we need to flip T_{i+1} if that index exists or else we can flip T_i .

EXPLANATION:

First let us solve the simpler version of the problem i.e, minimum number of operations to convert S into $1-$ *balanced string.* Then, if $S_i = 1$, we need to have either $S_{i-1} = 1$ or $S_{i+1} = 1$. To change S into a $1-$*balanced string*, we need to apply the following procedure:

- Iterate over i from 1 to N, suppose we encounter some $S_i = 1$, and no existing neighbour of it ($i - 1$ or $i + 1$) has a value equal to 1. Now we could either flip S_i or flip any of its neighbours. But it would be always optimal to flip S_{i+1} in this scenario because it would benefit another index $i + 2$ if it exists with value $S_{i+2} = 1$ and none of $i + 2$'s neighbours have value equal to 1. (For example to convert 0101 into $1-$ *balanced string*, it would be optimal convert the string to 0111 with 1 operation rather than converting it to 0000 using 2 operations) . If the index $i + 1$ doesn't exist we can flip the value S_i .

Now we know how to convert string S into a $1-$ balanced string . But how do we convert into a $K-$ balanced string ? The key idea lies in the fact of splitting string into and array $vals$ of K strings where $vals_i = S_i S_{i+k} S_{i+2k} \ldots$ for $0 \leq i < K$. The reason we want to split is that if we look at some index i, it is only interacted by indices $i - k$ and $i + k$.

Now the problem reduces to finding minimum number of operations to convert each string in $vals$ into a $1-$ *balanced string* and then summing those values.

TIME COMPLEXITY:

$O(N)$ for each testcase.

PROBLEM:

Chef is a very lazy person. Whatever work is supposed to be finished in x units of time, he finishes it in $m * x$ units of time. But there is always a limit to laziness, so he delays the work by at max d units of time. Given x, m, d find the maximum time taken by Chef to complete the work.

EXPLANATION:

As we can see the upper bound to complete the work is fixed, $i.e$ Chef cannot delay the work beyond this time. The upper bound for any work x is $x + d$ since he cannot delay any work by more than d units of time.

Also, it is given that any work that is assigned to Chef which is supposed to be finished in x units of time, he will finish that in $x * m$ units of time.

As we need to find the maximum time taken by Chef to complete the work, there will be two cases possible:

Case 1: $x + d \geq m * x$

- In this case, Chef can finish the work assigned to him in $m * x$ units of time, which is less than the upper bound of the time to finish the work. Hence the maximum time by Chef to complete the work, in this case, will be $m * x$.

Case 2: $x + d < m * x$

- In this case, the upper bound to finish the work by Chef is less than the maximum time he used to take to finish the work assigned to him. Hence, the maximum time Chef can take to complete the work assigned to him is $x + d$ in this case.

TIME COMPLEXITY:

O(1) per test case

PREREQUISITES:

Dynamic programming, Expected value, Modular multiplicative inverse

PROBLEM:

We are given an array of length N and it is shuffled. In each turn Alice picks an index which is not chosen yet and shows it to Bob. If the number in that index is larger than all the numbers taken till now then Bob adds that number to his array. We need to find out the expected length of Bob's array modulo 998244353.

QUICK EXPLANATION:

- Let us first sort the array. Let $dp[i]$ denote the expected length of Bob's array if the first chosen index is i.
- This can be handled in two cases. If $A_i = A_i + 1$, then $dp[i] = dp[i+1]$ else $dp[i] = 1 + \frac{dp[i+1]+dp[i+2]+\cdots+dp[N]}{N-i}$.
- Since we can pick every starting index i with probability $\frac{1}{N}$, the final answer will be $\frac{dp[1]+dp[2]+\cdots+dp[N]}{N}$.

EXPLANATION:

First let us sort the array as it makes our approach simpler and doesn't change the final solution of the problem. This problem can be solved by using dynamic programming. For this we have to define a dp state. Let $dp[i]$ denote the expected length of Bob's array if the first chosen index is i.

The base case is $dp[N] = 1$ since all the remaining elements are $\leq A_N$ and thus cannot be further added to Bob's array.

Now let us iterate from $N-1$ to 1 and calculate the dp states.

Let us suppose we are at some index i. Now we need to handle the following two cases:

Case 1: $A_i = A_{i+1}$

- In this case, the number of elements greater than A_i is the same as that for A_{i+1} and thus the possible final Bob's array states must also be same. Therefore, $dp_i = dp_{i+1}$.

Case 2: $A_i \neq A_{i+1}$

- In this case suppose the first element we pick is A_i. The number of elements greater than A_i is $N-i$.
- Therefore, for the next element, we could pick index $i+1$ with probability $\frac{1}{N-i}$, index $i+2$ with probability $\frac{1}{N-i}$, $\ldots$, index N with probability $\frac{1}{N-i}$.
- Thus, $dp[i] = 1 + \frac{dp[i+1]+dp[i+2]+\cdots+dp[N]}{N-i}$.

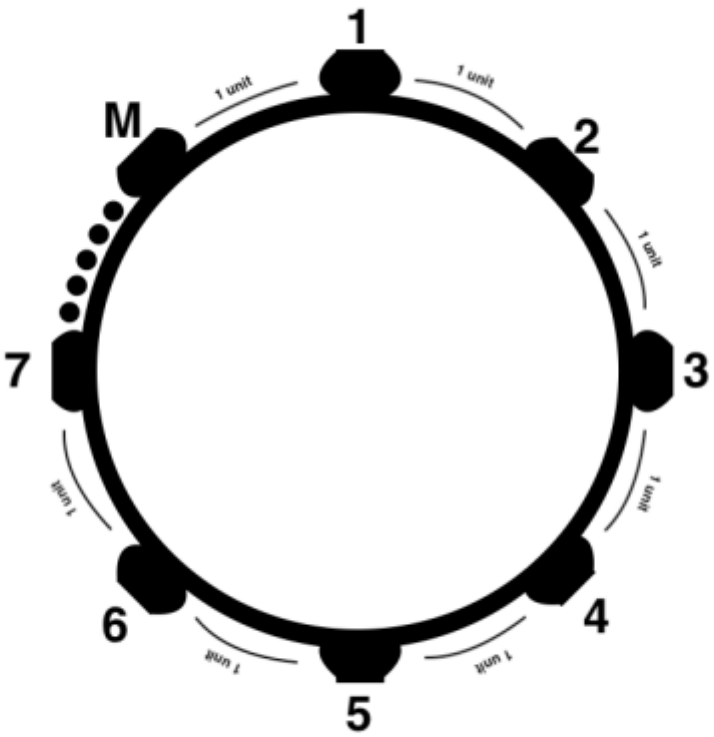
Once we have our dp values, our final answer will be $\frac{dp[1]+dp[2]+\cdots+dp[N]}{N}$. This is because we can pick the starting index as index 1 with probability $\frac{1}{N}$, index 2 with probability $\frac{1}{N}$, $\ldots$, index N with probability $\frac{1}{N}$.

TIME COMPLEXITY:

$O(N \log N)$ or $O(N \log N + N \log MOD)$ per testcase depending on the implementation for calculating modular multiplicative inverse.

PROBLEM:

There is a circular track of length M consisting of M checkpoints and M **bidirectional** roads such that each road has a length of 1 unit.



Chef is currently at checkpoint A and wants to reach checkpoint B. Find the **minimum** length of the road he needs to travel.

EXPLANATION:

Let $A \leq B$ (otherwise we can swap them). Chef has two choices now:
First Choice: Start moving in clockwise direction until he reaches B. Distance travelled = $B - A$
Second Choice: Start moving in anti-clockwise direction until he reaches B. Distance travelled = $A + M - B$
Thus the answer is $min(B - A, A + M - B)$

TIME COMPLEXITY:

$O(1)$ for each test case.

PROBLEM:

Chef has a sequence of integers A of length N. He creates another sequence B of length $2 \cdot N$ using sequence A. Initially, B is empty. Chef performs the following process.

For each element A_i $(1 \leq i \leq N)$ of A:

- Choose any arbitrary integer k (Note that the value of k can be different for different elements).
- Add $A_i - k$ and $A_i + k$ to B.

Chef then shuffles the sequence B randomly after this process.

You're provided with a sequence B of size $2 \cdot N$. You are required to tell if the provided sequence can be achieved from any sequence A of size N using the given process or not.

QUICK EXPLANATION:

- The answer is `YES` only if the sum of elements of B is **even**.

EXPLANATION:

Observation

Let us suppose we have two integers X and Y. We need to check if there exists an integer Z such that $X = Z - k$ and $Y = Z + k$.

Assuming $X = Z - k$ and $Y = Z + k$, the value of $X + Y$ would be $2 \cdot Z$. Thus, $Z = \frac{X+Y}{2}$.
In conclusion, if the value $X + Y$ is even, there exists an integer Z such that $X = Z - k$ and $Y = Z + k$.

Based on above observation, for an array A (of size N) to exist, we need to divide array B (of size $2 \cdot N$) into N pairs such that:

- Each element of B is present in **exactly** one pair.
- The sum of each pair is **even**.

If we are able to achieve this, then, we have a possible array A. The elements of array A would be equal to the mean of the pairs formed by elements of array B.

Claim: There exists a possible array A if the sum of the elements of B is even.
Proof: Let X denote the count of odd elements in B and Y denote the count of even elements in B.

- Since the sum of all elements of B is even, X (count of odd elements) must be even.
- Total count of elements in B is $2 \cdot N$ (which is even) and the count of odd elements is also even. Thus, the count of even elements, $Y = (2 \cdot N - X)$ is even.

For the Y even elements, we can divide them into $\frac{Y}{2}$ pairs. Each pair contains 2 even elements. Thus, each of these pairs has an even sum and is a valid pair.
For the X odd elements, we can divide them into $\frac{X}{2}$ pairs. Each pair contains 2 odd elements. Thus, each of these pairs has an even sum and is a valid pair.

Hence proved that we can construct an array A from B if the sum of the elements of B is even.

In conclusion, we only need to check the sum of the elements of B. If the sum of elements is even, the answer is `YES`, else it is `NO`.

PREREQUISITES:

Combinatorics, Dynamic Programming, Sliding Window

PROBLEM:

Given a permutation array of length N. Determine the number of ways to assign a direction (left or right) to each number such that it is possible to sort the permutation by applying the following operation any number of times - Select any element of the permutation and move it to the left/right end of the array, based on the assigned direction.

EXPLANATION:

Let $f(i)$ represent the number of *good* assignments, where the integers (**NOT indices**) $1, 2, \ldots, i$ are assigned direction `left` and integer $i+1$ is assigned direction `right`.
Then, the answer is equal to $f(0) + f(1) + \cdots + f(N)$.

Let us determine $f(i)$ for a fixed i.
Since we assign the first i integers direction `left`, we can sort them by moving them to the left in increasing order.

Then, we only concern ourselves with sorting the remaining integers.
Let $pos[x]$ represent the index of integer x in the permutation array.

Hint 1

If $pos[i+2] < pos[i+1]$, integer $(i+2)$ must be assigned direction `right` (Why?)

Generalising, if for any x, $pos[x+1] < pos[x]$ and integer x is assigned direction `right`, all integers j $(j > x)$ must be assigned direction `right` (the reasoning is similar as above and easily deducible).

Hint 2

If $pos[i+1] < pos[i+2]$, integer $(i+2)$ can be assigned either `left` or `right`.

Why?

On extrapolating, it is easy to see that, if $pos[i+1] < pos[i+2] < \cdots < pos[i+k]$, then all integers $(i+1), (i+2), \ldots, (i+k)$ will never have to be operated on, and hence can each have their direction set to either `left` or `right`.

Solution

TIME COMPLEXITY:

Calculating array S can be done using sliding window in $O(N)$. Computing $f(i)$ can be done in $O(1)$ by precomputing the powers of 2. Since we calculate $f(i)$ for each valid i, the total time complexity is

$$O(N+N) \approx O(N)$$

per test case.

PREREQUISITES:

GCD, Binomial Coefficients, Euler totient function

PROBLEM:

Once in the magic city of Lulympus, students were preparing for the final stage of Lulympiad, the national team selection for the IOI. After solving 3SAT in polynomial time complexity, learning the 4D convex hull trick and many other trivial techniques, Chef Lulus AKed the "Baraj Seniori" and decided to take a break and solve the following trivial problem:

You are given a positive integer N. Let P be the set of all permutations of length N. Find the value of the following expression: $\sum_{p \in P} \sum_{i=1}^{N} \gcd(p_1, p_2, \ldots, p_i)$.

Here $\gcd(p_1, p_2, \ldots, p_i)$ denotes their greatest common divisor.

Since the answer can be large, print it modulo MOD.

QUICK EXPLANATION:

Fix the length of the prefix of the permutation and then calculate the number of times a particular gcd value g occurs for this length and multiply it with g. Calculate this for all prefix lengths and output the sum.

EXPLANATION:

$O(Nlog^2(N))$ approach

Let i be a particular length of prefix of the permutation and g be a possible gcd value for this prefix length. This means that all element values from index 1 to index i are divisible by g.

$\because N/g \geq i$

$\therefore N/i \geq g$

For a prefix length i the possible value of gcd g goes from 1 upto N/i. Take any i elements from N/g elements which are divisible by g and rearrange them in the prefix and rearrange other elements in the suffix to obtain the total number of prefixes such that the gcd of the prefix of length i is divisible by g. To calculate the number of prefixes of length i such that the gcd is exactly g we need to subtract the number of prefixes of length i such that the gcd is exactly $2 \cdot g, 3 \cdot g, ...$ and so on from the above calculated permutations. Then multiply this number by g and add it to the $final\ ans$.

Time Complexity

We run a loop for each prefix length from $i = 1$ to N and inside each loop we iterate gcd values from $g = N/i$ to 1 in decreasing order. To remove overcounting we now run a loop from $2 \cdot g$ to N/i.

This totals to $\sum_{i=1}^{N} \sum_{g=N/i}^{1} N/(i \cdot g) \leq Nlog(N) \sum_{i=1}^{N} 1/i \leq Nlog^2(N)$

$O(Nlog(N))$ approach

The core idea of the problem is still same but now to avoid overcounting we won't run a loop from $2 \cdot g, 3 \cdot g, ...$ instead we will use the property of Euler's totient function:

$\sum_{d|n} \phi(d) = n$

Here the sum is over all positive divisors d of n.

For instance the divisors of 10 are $1, 2, 5\ and\ 10$. $\phi(1) + \phi(2) + \phi(5) + \phi(10) = 1 + 1 + 4 + 4 = 10$

In the previous approach we multiplied the the number of prefixes of length i such that the gcd is exactly g by g. Here we will multiply the number of prefixes of length i such that the gcd is divisible g by $\phi(g)$. Let the number of prefixes of length i such that the gcd is exactly g be a then we need to add $a \cdot g$ into our answer instead we have added $a \cdot \phi(g)$ into our answer. If we observe carefully we can see that these a prefixes will be repeated in the answer for every divisor of g which in the end would add upto $a \cdot (\sum_{d|g} \phi(d) = g)$

Time Complexity

We run a loop for each prefix length from $i = 1$ to N and inside each loop we iterate gcd values from $g = N/i$ to 1 in decreasing order. Euler phi values are precalculated.

This totals to $\sum_{i=1}^{N} N/i \leq Nlog(N)$

TIME COMPLEXITY:

$O(Nlog^2(N))$ or $O(Nlog(N))$ for each test case.

PROBLEM:

You are standing in front of a hallway with N consecutive doors. You want to cross through the N doors.

You may only walk through open doors. You also have a magic wand, that can flip the state of all doors. Determine the minimum number of times you need to use the wand.

EXPLANATION:

The solution is rather straightforward.

If the door we have to cross is open, simply walk through it (it makes no sense to use the wand and close the door) and if the door is closed, use the wand, flipping the status of all doors, and then walk through the door.

All that remains is to implement the above idea efficiently. Everytime we use the wand, it is inefficient to manually flip the status of all doors.

Rather, if we have used the wand k times thus far, the status of any door is equal to its initial state if k is even, and is flipped otherwise (the proof of which is trivial and left to the reader as an exercise).

Therefore, we iterate through the doors from left to right, and in each step, check if the door is open or closed (with the above trick) and use the wand appropriately.

TIME COMPLEXITY:

Since we iterate through the N doors once, the total time complexity is

$$O(N)$$

per test case.

PROBLEM

Chef has an array A having N elements. Chef wants to make all the elements of the array equal by repeatedly using the following move.

- Choose any integer K between 1 and N (inclusive). Choose K distinct indices $i_1, i_2, \cdot, i_K$, and increase the values of $A_{i_1}, A_{i_2}, \cdots, A_{i_K}$ by 1.

Find the minimum number of moves required to make all the elements of the array equal.

QUICK EXPLANATION

The number of operations needed is $max(A_i) - min(A_i)$

EXPLANATION

We can pick any subset of given integers and increase elements in the chosen subset by 1. We want to make all elements equal.

One thing we can see is that we can only increase the elements. So it doesn't make sense to include the largest element in any operation.

Let us perform the operations in the following manner. In current operations, pick all indices x such that A_x is the minimum element. This way, after the operation is performed, the minimum of A has increased by 1.

Let's assume minimum of A is $min(A_i)$ and maximum of A is $max(A_i)$. In one operation, we can increase the minimum of A by 1. All the elements would be equal when $min(A_i) = max(A_i)$. So it requires $max(A_i) - min(A_i)$ operations.

Hence, all we need to do is to compute the minimum and maximum of given A and print their difference.

TIME COMPLEXITY

The time complexity is $O(N)$ per test case.

PROBLEM

Given two integers N and S, Find maximum possible value of $|T_1 - T_2|$ if $0 \leq T_1, T_2 \leq N$ and $T_1 + T_2 = S$

It is guaranteed that for N and S in input, there exists at least one pair (T_1, T_2) such that $0 \leq T_1, T_2 \leq N$ and $T_1 + T_2 = S$

QUICK EXPLANATION

- If $S \leq N$, we can choose pair $T_1 = 0$ and $T_2 = S$ to obtain absolute difference S, which is maximum possible.
- If $N \leq S \leq 2 * N$, we can choose pair $T_1 = N$ and $T_2 = S - N$ to obtain absolute difference $2 * N - S$
- There cannot be a case with $S > 2 * N$ as that requires at least one of T_1 and T_2 to be $> N$ which is not allowed. Similarly for $S < 0$.

EXPLANATION

I'd explain two thought processes here.

Thought Process 1

Let's assume we only require $0 < T_1, T_2$ and $T_1 + T_2 = S$. (Upper bound N is ignored). It is easy to see that if we choose $T_1 = 0$ and $T_2 = S$, we obtain maximum possible absolute difference. Increasing T_1 only decreases T_2 which reduces absolute difference. So we achieve absolute difference S.

In our problem, $T_1, T_2 \leq N$ stops us from choosing above, as it may happen that $T_2 = S > N$. So we have to reduce T_2 by at least $S - N$. Let's do that. $T_1 = S - N$ and $T_2 = N$ is the pair we get, and we don't need to do any more changes, as reducing T_2 now only reduces absolute difference. So, we obtain an absolute difference $2 * N - S$.

Thought Process 2

In most min-max problems, it is always good to check out boundary points. We can prove that the absolute difference is maximized only when at least one of T_1 and T_2 are on end-points (0 or N).

Based on the above, we can try all pairs where T_1 is either 0 or N (There are only two such pairs). If a valid pair is formed, then take the maximum of absolute difference among valid pairs.

Tip

The required answer is $min(S, 2 * N - S)$. The proof is left as an exercise.

TIME COMPLEXITY

The time complexity is $O(1)$ per test case.

PROBLEM:

Value of an array A of length L is defined as the sum of $(A_i \oplus i)$ for all $0 \leq i < L$, where $\oplus$ denotes bitwise xor operation. **Note** that array indices start from zero.

You are given an integer N. Then you create an array A consisting of 2^N integers where $A_i = i$ for all $0 \leq i < 2^N$.

You can do at most K operations on this array. In one operation, you can choose two indices i and j $(0 \leq i,j < 2^N)$ and swap A_i and A_j.

What is the maximum value of array A you can obtain after at most K operations?

EXPLANATION

Initially $A_i = i$ hence $A_i \oplus i = 0$ for all $0 \leq i < 2^N$. Here we can get max xor $A_i \oplus i = 2^N - 1$ by setting $A_i = i \oplus (2^N - 1)$. In one operation we are swapping two elements so we can make two elements equal to $2^N - 1$. So best way is to swap A_i with $A_i \oplus (2^N - 1)$.

So we just need to find out how many elements we would be able to make equal to $2^N - 1$ in K moves. In one move, we can make two elements, hence in K moves at most $2 \cdot K$. But also there are only 2^N elements hence take the minimum i.e. $min(2 \cdot K, 2^N)$.

After these operations if i^{th} element has become $A_i = i \oplus (2^N - 1)$ then it would contribute $2^N - 1$ to the value because $A_i \oplus i = i \oplus (2^N - 1) \oplus i = 2^N - 1$. And all other elements would contribute 0 as they are not changed. Hence answer is $min(2 \cdot K, 2^N) \times (2^N - 1)$.

TIME COMPLEXITY:

$O(N)$ per test case

PROBLEM:

Given two arrays A and B each of length N.

Find the value $\sum_{i=1}^{N} \sum_{j=1}^{N} \max(A_i \oplus B_j, A_i \& B_j)$.
Here $\oplus$ and $\&$ denote the **bitwise XOR operation** and **bitwise AND operation** respectively.

Quick Explanation:

First of all let us see that pattern in $max(x \oplus y, x \& y)$.
If the MSB of x and y are i and j respectively, then ,

- if $i \neq j$ then $x \oplus y > x \& y$.
- if $i = j$, then $x \oplus y < x \& y$. This is because maximum of both will be the one with highest bit ON.

Now, for array b store the following for each i, sum of all values of b that has i as its MSB. Now, iterate through a[i] and add contribution according to its MSB to the sum.

Explanation

We will define arrays to store different values as follows:
$msb[i] \rightarrow$ number of elements in a having their most significant bit as i.
$bits[i] \rightarrow$ number of elements in a having their i_{th} bit on.

$bit\_table[i][j] \rightarrow$ number of elements in a having their most significant bit as i, with j_{th} bit on.

We will calculate values for these arrays using simple bit manipulation and then proceed to calculate the sum as follows.

We will loop through the elements of B and see for each element as B_i, having it most significant bit as k.

- If all bits of B_i is unset then we would simply take the sum of array A and add it to our answer, since $A_j \oplus B_i = A_j, 0 \leq j < n$
- Otherwise we would loop through the bits of B_i and for each bit say j
  - if j_{th} bit is set, then we add the following to our final answer:

$$answer+ = (1 << j) \times bit_table[k][j]$$

$$answer+ = (1 << j) * (n - msb[k] - (bits[j] - bit\_table[k][j]))$$

- if j_{th} bit is not set, then we add the following:

$$answer+ = (1 << j) \times (bits[j] - bit\_table[k][j])$$

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES

Euler's totient function

PROBLEM

Given two integers a and b, we define $f(a, b)$ as the maximum value of $|\gcd(a, x) - \gcd(b, x)|$ where x is some natural number. Formally,

$$f(a, b) = \max_{x \in \mathrm{N}} |\gcd(a, x) - \gcd(b, x)|$$

(where N represents the set of natural numbers and $\gcd(a, b)$ represents the greatest common divisor of a and b)

You are given an integer k. You need to find the number of ordered pairs (a, b) such that $f(a, b) = k$.

QUICK EXPLANATION

- A pair (a, b) shall have $f(a, b) = (max(a, b)/g - 1) * g$ where $g = gcd(a, b)$
- The number of pairs with $f(a, b) = k$ shall have

EXPLANATION

Rewriting $f(a, b)$

Let's consider a pair (a, b) with $a < b$ and try to compute $f(a, b)$. We want to find the maximum difference between $gcd(a, x)$ and $gcd(b, x)$. Sinec $1 \leq gcd(p, q) \leq p, q$ for any positive integers p and q, the maximum difference we can achieve is $b - 1$, wheren $gcd(b, x) = b$ and $gcd(a, x) = 1$

But this can happen only when $gcd(a, b) = 1$. Hence, for pairs (a, b) where a and b are co-prime, $f(a, b) = max(a, b) - 1$.

Let's generalize. Considering a pair (a, b) such that $gcd(a, b) = g$, then we can see that $f(a/g, b/g) = max(a, b)/g - 1$ and $f(a, b) = g * f(a/g, b/g) = g * (max(a, b)/g - 1) = max(a, b) - g$.

Hence, we can write $f(a, b) = max(a, b) - gcd(a, b)$.

Additional observation is that $f(a, a) = a - a = 0$, So they do not affect our answer. Hence, we can consider only unordered pairs (a, b) where $a < b$ and then multiply the answer by 2. So, For the rest of the editorial, assume $a < b$.

The number of pairs with $f(a, b) = K$

We need number of pairs (a, b) where $a < b$ such that $b - gcd(a, b) = K$. Assuming $g = gcd(a, b)$, g must divide K as well.

Let's substitute $a = p * g$ and $b = q * g$. So we have $q * g - g = K$. We can see that p and q must be co-prime.

Hence, let's iterate over all factors of K. If the current factor is g, we get $q - 1 = K/g$. So we must have $q = 1 + K/g$. $1 \leq p < q$ and $gcd(p, q) = 1$. What is the number of values of p satisfying these conditions?

Let's define function $\phi(N)$ as the number of integers co-prime to N up to N. i.e. the number of integers x such that $1 \leq x \leq N$ and $gcd(x, N) = 1$.

Hence, for each factor g of K, we have $\phi(K/g + 1)$ choices of a and exactly one choice for b, contributing $\phi(K/g + 1)$ pairs to the number of pairs (a, b) with $f(a, b) = K$

The $\phi(N)$ function

The $\phi(N)$ function is well known as the Euler totient function, which can be computed beforehand for all integers from 1 to $M = 10^6 + 1$ in $O(M * log(log(M)))$ time in sieve style, as explained in **this** article.

It is important to compute $\phi(N)$ for $10^6 + 1$ as well, as when we consider $K = 1000000$ and $g = 1$,we need $\phi(1000001)$.

Following pseudocode represent precomputing all answers from 1 to M assuming $\phi(N)$ is already calculated.

```
for g in 1 to M:
    for k = g to M, step size g:
        ans[k] += phi[k/g+1]
```

TIME COMPLEXITY

The time complexity of the above solution is $O(M * log(M) + T)$, though the time limit was relaxed enough to allow certain $T * \sqrt{M}$ kind of approaches to pass.

PREREQUISITES:

Dynamic Programming

PROBLEM:

You are given two arrays A and B, both of length N.

You would like to choose **exactly** K **distinct** indices $i_1, i_2, \ldots, i_K$ such that $\min(A_{i_1} + A_{i_2} + \ldots + A_{i_K}, B_{i_1} + B_{i_2} + \ldots + B_{i_K})$ is **maximized**. Find this maximum value.

QUICK EXPLANATION:

What all information do we need to describe a state of the problem?

Suppose we have seen first i indices. We will require the following information to describe the state:

- The number of indices j that we have chosen till now.
- Possible sum of values of array A for the chosen indices - let's call it Sum_A
- Possible sum of values of array B for the chosen indices - let's call it Sum_B

Interesting values of Sum_B, for a given Sum_A!

Let us analyze the state where will have seen first i indices, and have selected j indices out of the first i indices. Also, let say that one of the possible values of Sum_A is S_A.

Now for this value S_A there can be several possible values of Sum_B. Which of these values is interesting for our problem?

The answer is, the maximum possible value of Sum_B is the only value which is required to solve our problem!

The Final DP

Based on the above points, let us have $dp[i][j][s]$, where i represents the number of indices that we have seen so far, j represents the number of indices that we have selected so far, and s represents the sum of values of array A for the chosen indices. The value of $dp[i][j][s]$ represents the the maximum sum of values of array B possible when the sum of values of array A is s.

We can optimize the memory usage just like we do in the classical knapsack. We can remove the state i, and have $dp[j][s]$, and update it as we iterate over i.

EXPLANATION:

So let us first start by counting the number of ways in which we can choose K elements, out of the N elements. We know that it is $\binom{N}{K}$. To get an estimate of it's value, let us substitute $N = 40$, and $K = 20$. The number of ways turns out to be 1.38×10^{11}, which is very large. So, we can't try all possible ways and get our optimal answer!

In the problems which involves choosing exactly K elements out of N, and maximizing or minimizing something, it usually helps to think in the direction of Dynamic Programming.

A useful way to start thinking is, what happens when I have seen first i elements, and I have selected j elements out of them. What other information is required to completely describe this state?

In our current problem, we will need the list of all possible values of Sum_A and the corresponding Sum_B, where Sum_A represents the sum of values of A for the chosen j indices (and similarly for Sum_B)

Once we have all these information for all possible states, we can get our answer by iterating over all possible values of Sum_A and Sum_B when $i = N$, and $j = K$. However, calculating these value is not efficient, as it can take $(40 \cdot 40 \cdot 1600 \cdot 1600 \approx 4 \cdot 10^9)$ instructions for each test case, which is not feasible.

So, can we somehow reduce the possible number of states, and focus on the states which are *interesting* for us? Let us fix the values of i, j, Sum_A, and focus on all the possible values of Sum_B. Suppose there are two values S_1 and S_2, such that $S_1 < S_2$. Intuitively, it looks like we should always take S_2 for further discussions, because if $Sum_A < S_1$ and $Sum_A < S_2$, then S_2 is better, otherwise, Sum_A will be driving the answer, and therefore we can maximize Sum_B. This is just an intuitive statement. To prove it mathematically, we can make cases where

- $S_1 < Sum_A$ and $S_2 < Sum_A$
- $S_1 < Sum_A$ and $S_2 > Sum_A$
- $S_1 > Sum_A$ and $S_2 > Sum_A$

and further analyzing with the new upcoming values of A_{i+1} and B_{i+1}. After this exercise, we can see that it is always better to choose maximum value of Sum_B.

Now, after this realization, we can have $dp[i][j][s]$, where i represents the number of indices that we have seen so far, j represents the number of indices that we have selected so far, and s represents the sum of values of array A for the chosen indices. The value of $dp[i][j][s]$ represents the the maximum sum of values of array B possible when the sum of values of array A is s.
We can optimize the memory usage just like we do in the classical knapsack. We can remove the state i, and have $dp[j][s]$, and update it as we iterate over i.

So if we know value of $dp[i][j][s]$, and now I take index i, I will be able to update the value of -
$dp[i+1][j+1][s+A_{i+1}] = max(dp[i+1][j+1][s+A_{i+1}], dp[i][j][s] + B_{i+1})$ and
$dp[i+1][j][s] = max(dp[i+1][j][s], dp[i][j][s])$

To finally get our answer, we can iterate over all the possible values of s when $i = N$ and $j = K$, and find the maximum value of $min(s, dp[N][K][s])$

TIME COMPLEXITY:

Sum_A and Sum_B are bounded by $Max_{A_i} \cdot N$.
We will iterate over i going from 1 to N, j going from 1 to K, and Sum_A going from 1 to $Max_{A_i} \cdot N$. Hence, Time Complexity per test case will be $O(N^2 \cdot K \cdot Max_{A_i})$

PREREQUISITES:

Bitwise Operations

PROBLEM:

Alice and Bob are ready to play a new game. Both the players take alternate turns. Alice starts first.

There are N **binary** numbers written on a blackboard.

- Alice, in her turn, erases any 2 numbers from the blackboard and writes the **bitwise OR** of those 2 numbers on the blackboard.
- Bob, in his turn, erases any 2 numbers from the blackboard and writes the **bitwise AND** of those 2 numbers on the blackboard.

Note that, after each move, the count of numbers written on the blackboard reduces by 1.

Both players play until a single number remains on the blackboard. Alice wants to **maximise** the remaining number while Bob wants to **minimise** the remaining number. Find the remaining number if both the players play optimally.

EXPLANATION:

Here we count the number of 1 and 0 written on the blackboard and store it in variables $ones$ and $zeroes$ respectively.
Now we observe that in one move Alice can take a 1 and 0 and remove the 0 from the board by taking bitwise OR while Bob can take a 0 and a 1 and remove the 1 by taking bitwise AND, provided both 1 and 0 exists.
In the end if only $1s$ exists then the last remaining would also be 1 only and same for 0.
Thus we just need to check the initial number of $1s$ and $0s$ and our answer would be based on whichever one is greater.

$$answer = \begin{cases} 1, & if\ ones \geq zeroes \\ 0 & if\ ones < zeroes \end{cases}$$

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Dynamic Programming

PROBLEM

There are N chapters and only K minutes are left before the exam. Chapter i will be worth M_i marks in the paper, and shall take T_i minutes to prepare. Chef can study the chapters in any order but need to study them completely and atmost once. He knows that he will forget the first chapter that he will prepare (due to lack of sleep) and won't be able to score any marks in it. Find the maximum marks that Chef can score in the exam.

EXPLANATION:

Since our goal is to maximize the Chef score, and we know that the Chef will forget the first chapter he prepares due to lack of sleep. So whatever the set of chapters Chef plans to prepare in given time, he will always prepare the chapter first which has minimum marks among the set of chapters he chooses.

We can use Dynamic Programming to find the maximum marks Chef can score in the given exam. Let's move towards it:

- $DP[0][j]$ denotes the maximum marks that Chef can prepare in j minutes such that he doesn't forgot any chapter.
- $DP[1][j]$ denotes the maximum marks that Chef can prepare in j minutes such that he will forgot exactly one chapter.

Suppose Chef wants to prepare the i^{th} chapter which takes t time to completely prepare it. So the maximum time Chef can start preparing this chapter such that he can prepare this chapter within the time is $(k-t)$. Hence Chef can start preparing i^{th} chapter from any time in $[0, k-t]$ endpoints inclusive.

Let j be the time Chef starts preparing for the i^{th} chapter which takes t time to prepare and carries m marks. Hence the transition will be as follows:

$$DP[0][j+t] = max(DP[0][j+t], m + DP[0][j])$$

and

$$DP[1][j+t] = max(DP[1][j+t], m + DP[1][j], DP[0][j])$$

Here $DP[0][j]$ is the maximum marks which Chef can score in j time when i^{th} chapter is not included yet and Chef doesn't forgets any chapter he prepares.

And, $DP[1][j]$ is the maximum marks which Chef can score in j time when i^{th} chapter is not included yet and Chef forgets the first chapter he prepares.

We need to carefully set the initial state of the $DP's$.

Finally, $DP[1][K]$ will be the maximum score that Chef can score in K minutes when he forgot the first chapter he prepares.

TIME COMPLEXITY:

$O(N * K)$ per test case

PREREQUISITES:

Greedy, Set or Map or Priority Queue

PROBLEM:

Given N different types of candies with price and sweetness. Now given Chef has D dollars and can take a maximum of 2 candies one in each hand, find the maximum total sweetness he can collect under the given constraints if each type of candy can be taken at most once.

EXPLANATION

Idea is to try every candy and pick best choice available for another candy and take maximum of sweetness. Bruteforce idea is to loop over all (i,j) pairs and see if they can be picked together i.e. if their price sum $<= D$. But this takes $O(N^2)$ time. We can optimise it to $O(NlogN)$. Lets see how!

First of all, lets make a vector of $\{price, sweetness\}$ pairs and sort it in ascending order of price. We are sorting so that we can process in order of price. For example if current price is P then just need to pick maximum sweetness candy out of those candies who price is less than or equal to $D - P$.

We can maintain a multiset S of candies which can be picked as alternatives and we can keep them sorted according to sweetness hence we can pick the most sweet candy easily as last item in multiset.

We can process from most priced candy to least priced. Initially S would be empty. Maintain a pointer i which denotes upto which index we have inserted in the set. Now if we are currently at index j then we know that if we move to $j - 1$, all the candies which were suitable option for j would also be suitable for $j - 1$ as $P_{j-1} \leq P_j$.
For current index j we can keep inserting i^{th} candy until we satisfy $i < j$ and $P_i + P_j \leq D$. Keep incrementing i by 1 until both conditions are satisfied.
Also if we had inserted upto index i and we come upto index $j \leq i$ traversing from back, we need to remove current i.e. j^{th} candy from S as we can not pick j^{th} candy twice.

If we select j^{th} candy then best choice among available options would be the last one in set S hence take the sum of sweetness of these 2 candies. If S is empty then just consider taking j^{th} candy alone. Final answer is maximum of this value among all j.

TIME COMPLEXITY:

$O(NlogN)$ per test case

PREREQUISITES:

Prefix Sum

PROBLEM:

You are given Chef's calendar for the next N days, defined as a binary string S of length N where $S_i = 0$ means that Chef has a holiday on the i^{th} day from now, and $S_i = 1$ means that Chef has to work on that day.

Chef wants to plan his vacations. For each vacation, Chef needs X consecutive holidays in his calendar. Obviously, he can only go on one vacation at a time.

Chef can take at most one extra holiday. That is, he can flip at most one digit in S from 1 to 0. If he does this optimally, what is the **maximum** number of vacations that he can go on?

EXPLANATION:

What if no extra holiday is allowed?

Consider a contiguous segment of $0's$ of length len. The number of times Chef can go to a vacation during this contiguous segment will be len/K.

So, if no extra holiday is allowed, we can first get the length of all contiguous segments of $0's$ in the string. The answer would be the sum of len/X for all len values that we encounter. Let's call this answer $orig\_ans$

What happens when a holiday is taken?

Let's say we take a holiday on i^{th} day. Then, the contiguous segment of $0's$ ending at index $i-1$ is merged with the contiguous segment of $0's$ starting at index $i+1$, with S_i also becoming 0.

So now, we need to account for these changes in the previous contiguous segments of $0's$ to get the maximum number of vacations that Chef can go to, when a holiday is taken on i^{th} day.

Calculating efficiently

The information that we need, when taking a holiday on i^{th} day, is the length of contiguous segment of $0's$ ending at index $i-1$ (let's say len_end_{i-1}) and the length of contiguous segment of $0's$ starting at index $i+1$ (let's say len_start_{i+1}).

Consider the original contiguous segments of $0's$. After this change, we now have removed contiguous segments of length len_end_{i-1} and len_start_{i+1}, and have added a new segment of length $len_i = len\_end_{i-1} + len\_start_{i+1} + 1$.

So the answer when a holiday is taken on i^{th} day, let say ans_i will be:
$ans_i = orig\_ans - \frac{len\_end_{i-1}}{K} - \frac{len\_start_{i+1}}{K} + \frac{len_i}{K}$.

To get our final answer, we need to take the max of $ans_i \ \forall i : S_i = 1$

So we can first calculate the length of contiguous segments of $0's$ that ends at index i, and length of contiguous segments of $0's$ that start at index i, for all $1 \le i \le N$. After this computation, we can efficiently calculate the maximum number of vacations that the Chef can go to, when the holiday is taken on i^{th} day, and finally take the maximum of all these numbers.

TIME COMPLEXITY:

The above approach will take $O(N)$ time for each test case.

PREREQUISITES:

Trie, Bitwise operations

PROBLEM:

You are given an array A of length N.

You can do the following operation on the array **atmost once**:

- Choose any **non-negative** integer X and a subarray $[L, R]$ $(1 \leq L \leq R \leq N)$ and update $A_i = A_i \& X$ for all $L \leq i \leq R$. Here $\&$ denotes bitwise AND operation.

Find the **maximum** bitwise XOR of **all** the elements of the updated array you can achieve.

EXPLANATION:

Let D be inital $bitwiseXOR$ of all the elements of the array A. It can be shown that all the bits in the binary representation of D set to 1 will always be set to 1 in the final answer.

Proof that the bits in D set to 1 will always be set to 1 in the final answer

Let us say that the i^{th} bit is set to 1 in D but it is set to 0 in the final answer.
$\implies$ The operation is done exactly once choosing an integer X such that i^{th} bit in X was 0. (as $(i^{th} bit)$ & $1 = i^{th} bit$ and this does not change the result).
Set i^{th} bit to 1 in X, this will set the i^{th} bit to 1 in the final answer as the count of i^{th} bits set to 1 won't change due to the operation.

This means we need to focus on the bits set to 0 in D and we want to change the parity of count of these to odd so that these can be included in the final answer.

Observation

$\&$ operation cannot set a bit to 1 from 0 it can only set it to 0 from 1 or leave it unchanged

We need to apply the operation on a subarray in a way such that $\sum_{i=0}^{30} 2^i$, where i represents the bits which have an odd count in the subarray, is maximum $i.e.$ the $bitwiseXOR$ of the subarray is maximum.

This problem is now deduced to a simpler problem : Find the maximum $bitwiseXOR$ of a subarray of A where we are concerned only with bits set to 0 in D.

Let P_i represent the $bitwiseXOR$ of elements in the prefix of length i of the array A.Take $bitwise\ OR$ of each element of P with D to remove the effect of the bits set to 1 in D.
The $bitwiseXOR$ of the subarray $[L, R]$ now is same as taking the XOR of P_R and P_{L-1}.

We need to find $max($ max $(XOR$ of two elements of the array P $), max(P_i))$. Let this value be Max.
Finding the maximum $bitwiseXOR$ of two numbers in an array is a standard problem which can easily be solved using a trie data structure in $O(Nlog(2^{30}))$.
Maximum XOR of Two Numbers in an Array
Thus, the final answer is $(D \mid Max)$
For details of implementation please refer to the attached solutions.

TIME COMPLEXITY:

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

PREREQUISITES:

Binary Search

PROBLEM:

There is an array A of size N. In one operation, we can increase the value of any of the element by 1. We need to find the minimum number of operations to perform on the array to make mean of the array equal to its median.

Median is defined as follows: Let B be the sorted array of A. if N is even, median is $B_{\frac{N}{2}}$ else the median is $B_{\frac{N+1}{2}}$.

QUICK EXPLANATION:

- Try binary searching on the final mean/median value of the array.
- For a given value, check how many operations are required to ensure mean = median = value.

EXPLANATION:

- Without loss of generality, let us assume N is odd.
- First let us sort the array A.
- Let us assume that in the optimal case, the final mean/median value be x. Let us say this requires y operations.
- We can clearly observe that it is always possible to achieve the final mean/median value as $x+1$ by adding 1 to each element of the array thereby requiring $y+N$ operations.
- If x can be achieved, then $x+1, x+2, \ldots$ can also be achieved with the number of operations increasing by N each time.
- Therefore, we can **binary search** on the **final mean/median value** and output the number of operations required to achieve that value.
- Now the only question remains is to given an x, find the number of operations to achieve mean = median = x.
- Let the sum of all the elements of the array A be sum.
- Let us find the number of operations to achieve mean = x. Now according to the conditions of mean, we have $\frac{sum+extra}{N} = x \implies extra = N \cdot x - sum$ where $extra$ is the total number of operations that should be performed.
- Let us also find the minimum number of operations required to achieve median = x. We can find this easily by ensuring the numbers in the indices from 1 to $\frac{N-1}{2} \leq x$ and the numbers in indices $\frac{N+1}{2}$ to $N \geq x$. Let this count be y.
- If $y > extra$, we cannot make mean equal to median since median requires atleast y operations. If $y \leq extra$, we can do $extra - y$ operations on A_N after making median equal to x. Since we are performing the remaining operations on the last element, without changing the median value, we are bringing the mean value to x.
- In this way, we can go on binary searching on the value of x and finally print the value of $extra$ at the minimum possible value of x for which we can make mean = median = x.

PREREQUISITES:

Dynamic Programming, Inversions

PROBLEM:

You have two binary strings A and B, both of length N. You have to merge both the binary strings to form a new binary string C of length $2 \cdot N$. The relative order of characters in the original binary strings A and B should not change in the binary string C.

For example, if $A = 01011$ and $B = 10100$, one possible way to merge them to form C is: $C = 0101101100$.

Minimize the number of inversions in the merged binary string C.

As a reminder, a pair of indices (i,j) is an inversion for binary string C if and only if $1 \leq i < j \leq |C|$, $C_i = 1$ and $C_j = 0$.

QUICK EXPLANATION:

We will use the above notation of pair of indices (i,j) to denote inversions.

- Suppose you have already merged the substrings $A[1...i]$ and $B[1..j]$. So the next two characters that you need to consider are $A[i+1]$ and $B[j+1]$. Which character will you take next out of these two?
- Let $A[i+1]$ be 1, and we choose to take this character as the next character to merge. The **new inversions** that will be created in the future because of this step will be $\#0's$ in $A[i+1 \ ... \ N] + \#0's$ in $B[j+1 \ ... \ N]$.
- In other words, for these many indices j' in future, (i',j') will be an inversion pair, where $i' = i+j+1$, and $j' > i'$.
  The same argument holds for $B[j+1]$.
- Let $A[i+1]$ be 0, and we choose to take this character as the next character to merge. There will be no **new inversions** that will be created in the future because of this step (because $C_{i'}$ should be 1 for new inversions to be created in future that starts at this index).
- By above explanation, we can have a $dp[i][j]$ which denotes the number of inversions when $A[i...N]$ and $B[j...N]$ are merged. Note that it is little different than the above explanation, just to be consistent with how standard $DP's$ like LCS are defined. We Calculate $dp[i][j]$ in terms of $dp[i+1][j]$, $dp[i][j+1]$ and number of $0's$ in suffix of A and B.

EXPLANATION:

Such types of problems are usually based on Greedy or Dynamic Programming approaches.

We can start our analysis by considering the following situation. Let say we have already merged the substrings $A[1...i]$ and $B[1...j]$. So the next two characters that we need to consider are $A[i+1]$ and $B[j+1]$. The question is, which character should we choose?

Some greedy thinking can tell you that if either of one of $A[i+1]$ or $B[j+1]$ is 0, then we can take this character as our next character, as it will not increase the inversion count in future. However, if we have both $A[i+1]$ and $B[j+1]$ as 1, then we are stuck. Which 1 to take?

There can be some more greedy things that you can think of at this point, like maybe continue choosing those $1's$ which will lead to a 0 faster? But you will soon realize that all these greedy algorithms will fail.

This motivates us to think in the direction of Dynamic Programming. What happens when we choose, let say $A[i+1]$?
There are two main observations after this merge:

1. The **new inversions** that will be created in the future because of this step will be $\#0's$ in $A[i+1 \ldots N] + \#0's$ in $B[j+1 \ldots N]$.
2. We now have $A[i+2]$ and $B[j+1]$ as our next two characters, and we essentially have the same problem again, with the reduced strings $A[i+2 \ldots N]$ and $B[j+1 \ldots N]$!!

Same thing holds for $B[j+1]$.
If $A[i+1]$ is 0, and we merge $A[i+1]$, then we do not create any **new inversion**.

The above explanation suggests us to have a 2-dimensional DP, say $dp[i][j]$.
Let $cnt_a[i]$ stores the number of $0's$ in \$ in $A[i...N]$, and $cnt_b[i]$ stores the number of $0's$ in \$ in $B[i...N]$.

Let $dp[i][j]$ denote the number of inversions when $A[i...N]$ and $B[j...N]$ are merged.
Then we will have the following transitions:

If $A[i]$ is 0, and we merge this character, then $dp[i][j] = dp[i+1][j]$.
If $A[i]$ is 1, and we merge this character, then $dp[i][j] = dp[i+1][j] + cnt_a[i] + cnt_b[j]$.

Similarly for $B[j]$, and we take the minimum value of $dp[i][j]$ over all these 4 cases.

TIME COMPLEXITY:

Calculating the arrays cnt_a and cnt_b will take $O(N)$ time.
Calculating $dp[i][j]$ will take $O(1)$ time, provided that required dp values are calculates and stored.
So, the overall time complexity will be $O(N^2)$

PROBLEM:

You are given an array B containing N integers, each of which is either -1 or a non-negative integer.
Construct **any** integer array A of length N such that:

- $0 \leq A_i \leq 10^9$ for every $1 \leq i \leq N$
- If $B_i \geq 0$, A must satisfy $mex(A_1, A_2, \ldots, A_i)$=$B_i$
- Otherwise, $B_i = -1$, which means there is no constraint on $mex(A_1, A_2, \ldots, A_i)$

If there does not exist any array satisfying the constraints, print -1 instead.

Note: The mex of a set of non-negative integers is the smallest non-negative integer that does not belong to it.
For example,
$mex(1, 2, 3) = 0$, $mex(0, 2, 4) = 3$, $mex(0, 0, 0) = 1$

Quick Explanation

Take two sets x and y. x contains integers coming in B_i as positive integers and y contains the remaining integers from 0 to $N-1$.
Construct greedily from B_1 to B_n.
If B_i is -1 , print the elements from y set. else If B_i is positive , print the previous element of set x.
In this way we can construct valid array.
Note that it won't be possible to create array A if for any i we have $B_i \geq (i+1)$ or for any i, there is a B_j such that $0 \leq j < i$ and $B_j > B_i$

Explanation

There can be multiple ways to go about its implementation, one of which is discussed here.

First we check if for any i, if there exists any B_j such that $0 \leq j < i$ and $B_j > B_i$. If such B_j exists then it won't be possible to create array and we would return -1.

Now we would create two separate sets $slots$ and $elements$ where $slots$ represents the available slots of A that needs to be filled, while $elements$ represents the available elements that can be used to fill the slots.

Let's talk about the way we will go about filling. We will first remove duplicates from array B, i.e say array $B = \{-1, 1, 1, 1, 2\}$, then we would change duplicates to -1, so B would become $\{-1, 1, -1, -1, 2\}$. This way we would have increasing order of positive numbers and $-1s$. Now we will loop through array and upon reaching a positive value, we would loop through all available slots in A in increasing order and fill elements from set $elements$ till $Mex(A_1, A_2 .....A_i)$ becomes B_i. If for any case we do not have slots available to fill then also it won't be possible to create array A and we would return -1.

TIME COMPLEXITY:

In this algorithm, we iterate through the array and when we find any positive B_i, we loop through all the available slots available and remove them from set after filling it. Thus we are basically iterating the array twice and in second iteration of going through availble slots, we are accessing from a set that takes $O(logN)$ time, thus total time complexity would be $O(NlogN)$.

PREREQUISITES:

Prefix sums

PROBLEM:

We are given a string of size N consisting of only uppercase letters. A K - magical string is that string which has the length of longest non-decreasing subsequence equal to K . Given queries of the form L, R we need to determine if the substring from L to R can be converted into a K - magical string by rearranging the characters. If multiple such strings exist, we need to output the lexicographically largest one possible.

QUICK EXPLANATION:

- Let us calculate cnt_j which is the count of character j from L to R for every j from 1 to 26. This can be done by prefix sums.
- Let max be the maximum element in cnt.
- If $R - L + 1 < K$ or $max > K$, no answe is possible.
- Let T be the string in which all the characters in the positions from L to R are sorted in decreasing order.
- If $max = K$, we can directly output T as our answer. Else $max < K$, in this case we reverse the last K chracters in T and output the modified T as our answer.

EXPLANATION:

Let us initially preprocess the array $pref$ before answering queries. $pref_{i,j}$ denotes the total number of characters equal to j^{th} uppercase alphabet for all the indices from 1 to i.

If $R - L + 1 < K$, we clearly can't have an answer as the size of a $K-$ magical string must be atleast K.

Let us now consider $R - L + 1 \geq K$.

Let us calculate cnt using $pref$ where cnt_j is the total number of characters equal to j^{th} uppercase alphabet for all the indices from L to R. Let max be the maximum value in the cnt array.

Let us also consider string T which is all the character in the substring from L to R sorted in decreasing order.

Now consider the following three cases:

Case 1: $max > K$

- In this case, we cannot have an answer. This is beacuse we have an alphabet which is repeated more than K times. Then, the maximum non-decreasing subsequence of any rearranged string will be more than K because we always have a non-decreasing subsequence consisting of only this alphabet which has size $max > K$.

Case 2: $max = K$

- In this case, we can just output the lexicographically largest string possible i.e, T. This is because the maximum non-decreasing subsequence of T is the maxmimum count over all alphabets which is $max = K$.

Case 3: $max < K$

- In this case we need to modify T to increasing the maximum non-decreasing subsequence from max to K and also keep T as lexicographically large as possible. We can go about it by **simply reversing the last K positions of T.** Then, the last K positions are non-dereasing thus changing the maximum size of non-decreasing subsequence to K.

TIME COMPLEXITY:

$O(26 \cdot S + X)$ where S is the total number of queries overall all testcases and X is the the sum of $(R - L + 1)$ over all Q queries over all test cases.

PREREQUISITES:

Trees

PROBLEM:

Given a network of N cities (numbered from 1 to N) and $(N-1)$ roads arranged in a tree format with the root at city 1.
Each of the $(N-1)$ roads is assigned a value of either 0 or 1.

- If the value of a road is 0, it cannot be blocked by the government.
- If the value of a road is 1, it may or may not be blocked depending on the decision of the government.

Initially, city numbered 1 is infected by a virus. The infection spreads from an infected to an uninfected city only if **no** road present on the shortest path between the cities is blocked.

The government wants to control the number of infected cities and has asked you to devise a plan. Determine the **minimum** number of roads (with value 1) you need to block such that the **at most** K cities are infected at the end.
If it is not possible that **at most** K cities are infected, print -1.

EXPLANATION:

The first thing we need to calculate is all the subtrees that can be saved if we blocked entry in it. One thing to note here is if we can block entry to a subtree, say subtree T, then we won't consider the subtrees of T in our final count. We can use a DFS to calculate this.

Thus using DFS we will have an array A of our subtrees, where $A[i]$ denote the strength of the i_{th} subtree, i.e the number of nodes in the i_{th} subtree.
Initially we assume that no road is blocked, so we have n infected cities.
Now sort the array A in descending order and reduce the infected cities by $A[i]$ by blocking entry to the i_{th} subtree till the number of infected is greater than k.
If even after blocking all possible subtrees, the number of infected is greater than k, we give -1 as output, otherwise our final answer would be the number of subtrees that we blocked.

TIME COMPLEXITY:

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

PROBLEM:

There is a matrix of size $N \times M$. Two **distinct** cells of the matrix (x_1, y_1) and (x_2, y_2) are painted with colors 1 and 2 respectively.

You have to paint the **remaining** cells of the matrix such that:

- No two adjacent cells are painted in same color.
- Each color is an integer from 1 to 10^9 (both inclusive).
- The number of **distinct** colors used is **minimum** (including 1 and 2).

Note:

1. You cannot repaint the cells (x_1, y_1) and (x_2, y_2).
2. Two cells are adjacent if they share a common edge. Thus, for a cell (x, y) there are 4 adjacent cells. The adjacent cells are $(x, y+1)$, $(x, y-1)$, $(x+1, y)$, and $(x-1, y)$.

EXPLANATION:

Let us define the parity of a cell (x, y) as the parity of $x + y$. So, $(5, 1)$ is an even cell, whereas $(4, 1)$ is an odd cell. Consider a cell (x, y). Observe that if (x, y) is an even cell, then all the cells adjacent to (x, y) will be odd cells. Similarly, if (x, y) is an odd cell, then all the cells adjacent to (x, y) will be even cells. This property is useful while solving many grid related problems.

Consider the problem when none of the cells were already painted. In that case, we could have painted all the even cells by 1 and all the odd cells by 2. Note that the above coloring satisfies all the required properties.

Now if one of the cell (x, y) was already painted as 1, we could have assigned that color to all the cells having same parity as that of (x, y), and color 2 to the remaining opposite parity cells.

In the above problem, we are given two cells: (x_1, y_1) with color 1, and (x_2, y_2) with color 2. If the parity of both these cells are same, we can assign either color 1 or 2 to all the cells having same parity as these cells, and color 3 to all the remaining cells. We have used 3 distinct colors in this case.
If the parity of both these cells are different, we can assign color 1 to all the cells having same parity as that of (x_1, y_1), and color 2 to all the remaining cells (having same parity as that of (x_2, y_2)). We have used 2 distinct colors in this case.

TIME COMPLEXITY:

$O(N \cdot M)$ for each test case.

PREREQUISITES:

Permutations

PROBLEM:

You are given a set $\{0, 1, 2, \ldots, 2^N - 1\}$. Determine the number of permutations of the set such that, the maximum bitwise XOR of any even length sub array is minimised. Also output one such permutation.

EXPLANATION:

Define the *cost* of a permutation as the maximum bitwise XOR over all even length sub array's of the permutation.

Let X be the minimum possible cost over all permutations. Then, we are trying to find - (1) the number of permutations with cost X and, (2) output one such permutation.

Hint: Brute force helps. Try $N = 3$ and try finding X.

Explanation

Observe the pattern (derived using brute force code):

- $N = 1 \rightarrow X = 1$
- $N = 2 \rightarrow X = 2$
- $N = 3 \rightarrow X = 4$

It looks like $X = 2^{N-1}$, which is a good start.
We try proving this conjecture below.

Claim: $X = 2^{N-1}$ describes the relation between X and N.

Proof

Let's start by showing $X \not< 2^{N-1}$.
It is easy to show that, in every permutation, there exists some $a \geq 2^{N-1}$ and $b < 2^{N-1}$ adjacent to each other. And their bitwise XOR is always $\geq 2^{N-1}$. This implies $X \not< 2^{N-1}$.

A bit of experimentation gives a sequence with $X = 2^{N-1}$:

$$0, B + 0, B + 1, 1, 2, B + 2, B + 3, 3, \ldots$$

where $B = 2^{N-1}$. It's not hard to see that the bitwise XOR of any even length sub array is $\leq 2^{N-1}$. Thus, the claim stands proved.

Now, all we are left to find is the number of permutations whose cost equals $X = 2^{N-1}$ (an optimal permutation is presented in the above spoiler).

Claim: The cost of a permutation P is not affected by cyclic shifts of the permutation array.
(The proof is left to the reader as an exercise. Use the fact that (apart from the trivial $N = 1$) the XOR of all elements of the permutation is 0; thus the XOR of any sub array is equal to the XOR of the remaining elements in the array)

Represent a number as S if it is $< 2^{N-1}$, and B otherwise.
Claim: If a permutation has cost X, every S in it has an adjacent B.

Proof

We prove by contradiction.
Assume there exists a permutation with cost X, such that some cyclic shift of the it has a S with no adjacent B.
The permutation would look something like this:

$$\dots SSSB \dots$$

Since the cost of the permutation is X, the bitwise xor of the 3^{rd} and 4^{th} visible elements is bound to be 2^{N-1} (by the definition of S and B). And since the elements are distinct, the XOR of the 1^{st} and 2^{nd} elements is **strictly** between 0 and 2^{N-1}.
This implies that the XOR of the 4 elements is $> 2^{N-1}$, a contradiction (since the cost of the permutation is 2^{N-1}).

And we are done.

Claim: Every B should have **exactly one** S adjacent to it.

Proof

From the above claim, we know that each S should have at least one B adjacent to it. Since the number of S's is equal to the number of B's, each B should also have at least one S adjacent to it.

If a B has more than one S adjacent to it (as in $\dots S_1BS_2 \dots$), then either $S_1 \oplus B$ or $B \oplus S_2$ is $> 2^{N-1}$, breaking the optimality condition.
Thus proved.

From the two claims above, it isn't hard to deduce that the pattern should be of the form $\dots SBBSSBBS \dots$ where adjacent S and B are i and $i + 2^{N-1}$, respectively $(0 \leq i < 2^{N-1})$.

A bit of case work gives us a total of 4 *definite* patterns:

- $BBSSBBSS \dots$
- $BSSBBSSB \dots$
- $SSBBSSBB \dots$
- $SBBSSBBS \dots$

So, how many ways are there? Pairing each i and $i + 2^{N-1}$ gives us 2^{N-1} pairs, which can be permuted amongst themselves and arranged in one of the four patterns, giving us a grand total of $4 * (2^{N-1})!$ ways.

The case where $N = 1$ can be handled separately.

TIME COMPLEXITY:

Calculating the factorial in the answer takes $O(2^{N-1})$ time. Generating one required sequence takes $O(2^N)$.
The total complexity per test case is therefore:

$$O(2^{N-1} + 2^N) \approx O(2^N)$$

PREREQUISITES:

Bitwise operations

PROBLEM:

An array A of size N is called an interesting array if $N \geq 1$ and $A_i = 2^x - 1$ for some integer $1 \leq x \leq 60$. We need to find one such interesting array with minimum size where the bitwise xor of all the elements in the array is C.

EXPLANATION:

First let us analyse the binary representation of $2^x - 1$. It is $1111 \ldots$ (x times).

Therefore, intuition suggests us that this problem can be solved constructively as follows:

Let us take an initial empty array arr and also keep track of a variable cur which denotes the current xor of all the elements of arr.

(In the below method the word positon has 0-based indexing).

Now iterate over i from 60 to 0:

- If both bits in cur and C at position i are the same i.e either both are set or both are unset, we need not apply any operation and continue to next i.
- Else we need to add an element to the array in order to change to toggle the bit at position i for cur. Since we can add only numbers of the form $2^x - 1$, we will add $2^{i+1} - 1$ to arr and update cur as $cur = cur \oplus (2^{i+1} - 1)$.

After this entire procedure is done, there is one **special case** left to handle. What if $C = 0$? Then, we will end up with an empty array arr. But an interesting array must have size $N \geq 1$. Here, we can't have one element with $cur = 0$ since $2^x - 1 > 0$ for $1 \leq x \leq 60$. Therefore, in this special case, we need to add any two equal elements of the form $2^x - 1$, (for example $1, 1$) to arr.

Now, we can output the final array arr along with its size.

TIME COMPLEXITY:

$O(60)$ for each testcase.

PREREQUISITES:

Longest Increasing Subsequence

PROBLEM:

Chef received an array A of N integers as a valentine's day present. He wants to **maximize** the length of the **longest strictly increasing subsequence** by choosing a **subarray** and adding a fixed integer X to all its elements.

More formally, Chef wants to **maximize** the length of the **longest strictly increasing subsequence** of A by performing the following operation at most once:

- Pick 3 integers L, R and X $(1 \leq L \leq R \leq N,\ X \in \mathbb{Z})$ and assign $A_i := A_i + X\ \forall\ i \in [L, R]$.

Find the length of the longest **strictly** increasing sequence Chef can obtain after performing the operation **at most once**.

EXPLANATION:

Let say that the Chef has chosen L, R and X, and have assigned $A_i := A_i + X\ \forall\ i \in [L, R]$ to get the optimal answer. Also, assume that $X > 0$ for the current discussion. We will handle the other case later. So we can divide our array into three parts: $A[1 \ldots L-1], A[L \ldots R], A[R+1 \ldots N]$

Now, let say $B = \{A_{i_1} \ldots A_{i_x}, A_{j_1} \ldots A_{j_y}, A_{k_1} \ldots A_{k_z}\}$ be the final LIS, such that $i_u \in [1, L-1], j_u \in [L, R], k_u \in [R+1, N]$. Observe that if we further increase all A_i for $R < i \leq N$, the length of resulting LIS will either remain same as that of B, or it will increase. In simple words, increasing all A_i for $R < i \leq N$ only improves our solution.

So we can claim that if $X \geq 0$, we will only increase a suffix in the optimal case. If $X < 0$, we can have a similar analysis to show that we will only decrease the prefix, which is equivalent to increasing the remaining suffix by same amount, as long as LIS is considered. So, we only need to analyze $X \geq 0$, and $R = N$.

Let us now try to analyze what happens when we fix L. What is the maximum length of LIS that we can get. Now, we can divide our array into two parts: $A[1 \ldots L-1], A[L \ldots N]$.

Let $B_1 = \{A_{i_1} \ldots A_{i_x}\}$ and $B_2 = \{A_{j_1} \ldots A_{j_y}\}$ be the LIS of $A[1 \ldots L-1]$ and $A[L \ldots N]$ respectively. If we choose X in such a way that $A_{i_x} < A_{j_1} + X$, we can have our final LIS as $B_1 + B_2$, where the $+$ sign denotes the concatenation. Observe that this is the maximum length of LIS that we can get for this specific L.

Hence, for each i such that $1 \leq i \leq N$, we can first calculate $dp_1[i]$ = Length of LIS of $A[1 \ldots i]$ and $dp_2[i]$ = Length of LIS of $A[i \ldots N]$, and take the maximum of $dp_1[i] + dp_2[i+1]$ for all valid i.

TIME COMPLEXITY:

To calculate length of LIS for each i, we will require $O(N \cdot \log N)$ time. To further calculate answer, we will require $O(N)$ time. Hence the overall time complexity is $O(N \cdot \log N)$ for each test case.

PREREQUISITES:

Dynamic Programming

PROBLEM:

We are given an array A of N integers and Q ordered pairs of the form (x_i, y_i). We need to find the length of the longest magical subsequence of A and print any one of the subsequence indices. A subsequence of A say $(A_{i_1}, A_{i_2}, \dots A_{i_k})$ is said to be magical if for each $1 \leq j < k$, $(A_{i_j}, A_{i_{j+1}})$ must be equal to one of the Q pairs given.

EXPLANATION:

An $O(N \cdot Q)$ solution:

Let us initially forget about the time constraints and try to solve the problem. Let us try to solve this problem by using dynamic programming. For this, we need a state.

Let dp_i be defined as the length of longest subsequence ending at index i. We can initialize it to 1 initially. Now, dp_i is computed as follows: For every value x which forms one of the Q ordered pairs with A_i, update $dp_i = max(dp_i, dp_{recent_x} + 1)$ where $recent_x$ is a helper array which gives us the recent index inside the array A which has the value equal to x. This array $recent$ will be continuously updated as we iterate over i from 1 to N.

The computation of dp values is fairly straightforward. But there is only one problem, its time complexity. Clearly, the worst case time complexity of this approach is $O(N \cdot Q)$. Let us try to find a way to optimise it using square root idea.

An $O(N\sqrt{Q})$ solution:

Let us define $count(y)$ as the number of elements x for which (x, y) belongs to one of the Q ordered pairs.

Let us also define two groups big and $small$. Any number y belonging to big has $count(y) > \sqrt{Q}$ and any number y belonging to $small$ has $1 \leq count(y) \leq \sqrt{Q}$. These two groups can be computed in a straightforward manner. Clearly, the maximum number of values in big group possible will be $O(\sqrt{Q})$. (think why this is the case).

For the sake of reducing the time complexity, we need to define another array $dpVal$ where $dpVal_x$ is defined as the length of the longest subsequence ending at value x. Note that this array is different from dp since here I am talking about the actual values instead of the indices. Also, this array is intended only for the values in the $large$ group which we will understand why shortly.

The algorithm is now proceeded as follows:

Initialize $dpVal_x = 1$ for all values x belonging to the $large$ group.

Let us now iterate over i from 1 to N and for every iteration i :

- Initialize dp_i to 1.
- If the value at the current index A_i belongs to the $small$ group, then update dp_i by the steps mentioned in the first non-optimal solution. This takes $O(\sqrt{Q})$ time because of the definition of the $small$ group.
- If the value at the current index A_i belongs to the $large$ group, then update dp_i as $dpVal_{A_i}$.
- At the end of this iteration, we also need to update $dpVal$ for values in the $large$ group. For every value y of the $large$ group, if (A_i, y) belongs to the one of the Q ordered pairs, update $dpVal_x$ as $dpVal_x =$

$max(dpVal_x, dp_i + 1)$. This takes $O(\sqrt{Q})$ time since there are atmost $O(\sqrt{Q})$ values in the $large$ group.

Thus, by splitting the right hand side values of the Q pairs into $small$ and $large$ groups and cleverly utilizing them, we are able to reduce the time complexity of our solution from $O(N \cdot Q)$ to $O(N\sqrt{Q})$. Note that I haven't talked about constructing one such sequence which can be done easily by storing the last index information while calculating dp and $dpVal$ values.

You can refer the code for further understanding.

TIME COMPLEXITY:

$O(N\sqrt{Q})$ or $O(N\sqrt{Q} + N \log N)$ depending on the implementation.

PROBLEM:

An array is said to be *good* if all the elements of the array have the **same parity**. For example, the arrays $[2, 8]$ and $[9, 1, 3]$ are *good* while the array $[4, 5, 6]$ is not *good*.

Given an array A of size N. You can perform the following operation on the array:

- Choose indices i and j $(1 \leq i, j \leq |A|, i \neq j)$.
- Append $A_i + A_j$ to the array A.
- Remove the i^{th} and j^{th} elements from the array A.

Find the **minimum** number of operations required to make the array *good*.

It is guaranteed that the array can be converted to a *good* array using a finite number of operations.

EXPLANATION:

For a given array, if the existing number of odd numbers are odd then the resulting good array must have all odd numbers as the last odd number would always exist among the other even numbers. So the minimum chances to get to a point where there are no even numbers left in the array would be to use an odd number and an even number to get an odd number in every step. So, here the minimum steps would be the number of even numbers in the initial array.

If there are even number of odd numbers in the array, then the resuting good array may either have all numbers as odd or even. Let there be O odd numbers and E even numbers in the array. To make a good array of all even numbers, the way with least number of steps would be to add couple of odd numbers and make an even number, resulting in $O/2$ steps. Whereas, to make a good array with all odd numbers, each even number would be added to an odd number resulting in E steps. So in this case, the minimum steps would be $min(O/2, E)$.

TIME COMPLEXITY:

$O(N)$ for each test case as we traverse the array.

PROBLEM

Given an array A containing N integers, find the maximum value of $a * b + a - b$ where a, b are distinct elements of array.

QUICK EXPLANATION

- Rewriting $a * b + a - b = (a - 1) * (b + 1) + 1$, we can fix a and see that for b, we only need to consider either minimum or maximum of all elements of array excluding element a.
- It can be done by computing the minimum, second minimum, maximum and second maximum element of array, or just by simply sorting the array.

EXPLANATION

As mentioned in quick explanation, we can rewrite the expression.
$a * b + a - b = a * (b + 1) - (b + 1) + 1 = (a - 1) * (b + 1)$

Hence, we need to maximise $(a - 1) * (b + 1)$ where a and b are two distinct elements of array.

Now, a naive way to do this would be to try all pairs of numbers, and taking maximum of value resulted by any pair.

Here, if the elements of array were positive, Just taking the last two elements would have been sufficient. But there are negative elements too, and the product of two negative element can also lead to a higher positive product.

So we try to fix a. We can see that once a is fixed, it is optimal to choose b which would maximise value of $(a - 1) * (b + 1) + 1$. So

- if $(a - 1) > 0$, we'd aim to select the largest element in array, so that the value of $b + 1$ can be maximised, therefore $(a - 1) * (b + 1) + 1$ is maximised.
- if $(a - 1) < 0$, we'd aim to select the smallest element in array, so that the value of $b + 1$ can be minimised, therefore $(a - 1) * (b + 1) + 1$ is maximised.
- if $(a - 1) < 0$, then answer is atleast zero.

Hence, all we need to do is to sort the array, fix a to be each element one by one, and choose most optimum b. Taking maximum over all these candidates gives the maximum value of $a * b + a - b$

Bonus

Write a solution for above problem which checks minimum number of pairs.

TIME COMPLEXITY

The time complexity can be $O(N)$ or $O(N * log(N))$ depending upon implementation, per test case.
The memory complexity is $O(N)$ per test case, can be reduced to $O(1)$.

PREREQUISITES:

Strings

PROBLEM:

You are given two binary strings A and B, each of length N.

Consider a non-empty substring $A^{'}$ of A and a non-empty substring $B^{'}$ of B, both of the same length. Let $X = A^{'} \bigoplus B^{'}$ be the string obtained by taking the exclusive OR of $A^{'}$ and $B^{'}$.

You are also given a function $f(X)$, defined as

$$f(X) = \lfloor \frac{|X|}{2^{X_{10}}} \rfloor$$

where $|X|$ denotes the length of X and X_{10} denotes the decimal value of the binary integer represented by X. For example, if $X = 0110101$, then $|X| = 7$ and $X_{10} = 53$, so $f(X) = \lfloor 7/2^{53} \rfloor = 0$.

Your task is to **maximize** the value of the function $f(X)$ for the given strings A and B, by choosing $A^{'}$ and $B^{'}$ appropriately.

Note: The exclusive OR of two binary strings A and B such that $|A| = |B| = k$ is the unique binary string X of length k such that $X_i = 1$ **if and only if** $A_i \neq B_i$.

QUICK EXPLANATION:

In order to maximise the profit, we need to minimise the value of XOR as profit is inversely proportional to the XOR value. Also we need to maximise the length of the substring.
Therefore we just need to find the longest common substrings from both the strings S and Q as their XOR would be 0 and the length would be maximum. This would make the answer as $|X|$.

If there are no such substring, then the answer would always be 0

TIME COMPLEXITY:

$O(NlogN)$ for each test case.

PREREQUISITES:

Greatest Common Divisor

PROBLEM:

An array B of length N $(N \geq 2)$ is said to be good if the following conditions hold:

- For all $1 \leq i \leq N$, $2 \leq B_i \leq 10^6$
- $gcd(B_{i-1}, B_i) \neq 1$ for all i $(2 \leq i \leq N)$.

You have an array A of length N $(2 \leq A_i \leq 10^5)$. You want to make the array A good.

To do so, you can change **atmost** $\lceil \frac{2 \cdot N}{3} \rceil$ elements of A.

Print the final array after changing A to a good array. If there are multiple possible final arrays, **print any of them**.

It is guaranteed that A can be made good after changing **atmost** $\lceil \frac{2 \cdot N}{3} \rceil$ elements of A.

CORNER CASE:

If you are getting Wrong Answer, the following test case might help you:

Corner Case

$N = 6$
$A = \{P_1, P_2, P_3, P_4, P_5, P_6\}$ where $P_i > 10^4$, and each P_i is a prime number.

Make sure that your final array satisfies the condition $A_i \leq 10^6$, and you have not changed more than $\lceil \frac{2 \cdot N}{3} \rceil$ elements of A.

EXPLANATION:

This is one of the possible construction. Other solutions are also possible.

We are given that we can change **atmost** $\lceil \frac{2 \cdot N}{3} \rceil$ elements of A. Also note that the new elements should satisfy the following constraint: $2 \leq B_i \leq 10^6$.
One useful way to interpret it is: out of every 3 elements, we can change 2 of them, such that the conditions are satisfied. Let us try if we can achieve the above conditions through this interpretation.

Let us consider the first 4 elements of the array - A_1, A_2, A_3 and A_4 and let each of the A_i be a prime number for the sake of this discussion.

Now, because $gcd(A_1, A_2) = 1$, we would like to change one of the A_1 or A_2. Let us greedily choose A_2 and try to change it to some optimal value. Let $A_2 = \lambda \cdot A_1$, for some λ.

Again, we may have $gcd(A_2, A_3) = 1$ depending on the value of λ, so we need to suitably change A_3. Let the new value of A_3 be A_3' .

Because we want to change only 2 values out of every 3 values, we don't want to change our A_4. Hence, the following should also be satisfied: $gcd(A_3', A_4) \neq 1$. So, let us have $A_3' = \lambda' \cdot A_4$.

So, the current array looks like $\{A_1, \lambda \cdot A_1, \lambda' \cdot A_4, A_4, ...\}$. The only condition that now needs to be satisfied is $gcd(\lambda \cdot A_1, \lambda' \cdot A_4) \neq 1$.

Note that substituting $\lambda = \lambda' = 2$ will make the above property satisfied, as well as make sure that for each $A_i, 2 \leq A_i \leq 10^6$.

Hence, the final array would look like $\{A_1, 2 \cdot A_1, 2 \cdot A_4, A_4, 2 \cdot A_4, 2 \cdot A_7, A_7, ...\}$

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM

Chef's favorite game is chess. Today, he invented a new piece and wants to see its strengths.

From a cell (X, Y), the new piece can move to any cell of the chessboard such that its color is different from that of (X, Y).

The new piece is currently located at cell (A, B). Chef wants to calculate the minimum number of steps required to move it to (P, Q).

Note: A chessboard is an 8×8 square board, where the cell at the intersection of the i-th row and j-th column is denoted (i, j). Cell (i, j) is colored white if $i + j$ is even and black if $i + j$ is odd.

QUICK EXPLANATION

- If Chef is already at the target cell (P, Q), then 0 moves are required.
- Otherwise, if cells (A, B) and (P, Q) have different colors, then in one move, Chef can move directly from (A, B) to (P, Q).
- Otherwise, It requires the chef two moves, first to get to any opposite-colored cell, from which the chef can go to cell (P, Q)

EXPLANATION

The trivial case is when Chef is already at the target position, which happens when $(A, B) = (P, Q)$. No moves are required in this case.

Otherwise, at least one move is required. In one move, Chef can reach any cell with a color different from the color of cell (A, B). So if (P, Q) has a different color from cell (A, B), then Chef can reach the target in his first move.

Lastly, if the cell (P, Q) is of the same color as cell (A, B), the chef can in the first move, move to any cell with different color, from where the chef can move to (P, Q)

The following snippet covers the model solution.

```
read A, B, P, Q
if A == P and B == Q: print(0)
else if (A+B)%2 != (P+Q)%2: print(1)
else print(2)
```

TIME COMPLEXITY

The time complexity is $O(1)$ per test case.

PROBLEM:

You are given a binary string S of length N. You can perform the following operation on S:

Pick any set of indices such that no two picked indices are adjacent.
Flip the values at the picked indices (i.e. change 0 to 1 and 1 to 0).

For example, consider the string S = 1101101.
Pick the indices $\{1, 3, 6\}$.
Flipping the values at picked indices, we get $\underline{1}1\underline{0}11\underline{0}1 \rightarrow 0111111$.
Note that we cannot pick the set $\{2, 3, 5\}$ since 2 and 3 are adjacent indices.

Find the **minimum** number of operations required to convert **all** the characters of S to 0.

EXPLANATION:

For each test case we have a binary string of length N.

Now as *no adjacent bits can be flipped together*. Three cases are possible:

- When the string is already 0, then the number of minimum operations will be **zero**.
- When **no two adjacent bits are set**. In this case all the set bits can be converted to 0 in 1 operation.
- When the string contains adjacent set bits. In this case it can be shown that the answer will be 2 since all the odd-indexed set bits and even-indexed set bits can be identified as **two different sets**.

Evaluating all the above possibilities gives us the result.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

We are given a string of length N consisting of only numeric characters. We need to find the number of pairs i, j where $1 \leq i < j \leq N$ and $j - i = | S_j - S_i |$.

EXPLANATION:

- The main observation in this problem is to find the maximum possible value of $| S_j - S_i |$. The maximum possible value is 9 when $S_i = 0, S_j = 9$ or $S_i = 9, S_j = 0$.
- So, it is enough that we check for pairs i, j where $i < j$ and $j - i \leq 9$.
- We can theerefore iterate over i from 1 to N, and iterate over j from $i + 1$ to $\min(i + 9, N)$ and check the condition $j - i = | S_j - S_i |$ holds true or not. If it is true, we increment our answer.
- We will also fit our solution in the time limit because the number of pairs we are considering is $O(9 \cdot N)$ which takes linear amount of time to run.

TIME COMPLEXITY:

$O(N)$ for each testcase.

PREREQUISITES:

Binary Search, Digit DP

PROBLEM:

Find the minimum integer K such that sum of bits present on odd positions in binary representation of all integers from 1 to K is greater than or equal to N.

QUICK EXPLANATION

Use digit DP to find sum of bits present on odd positions in binary presentation of all integer from 1 to K for a particular K. As the sum would only increase with increasing K hence we can binary search on K.

EXPLANATION

Lets define a function $F(K)$ as sum of bits present on odd positions in binary representation of all integers between 1 to K. As sum would increase if we increase K hence
$F(K+1) \geq F(K)$

As F is a non-decreasing function, this gives an ideal condition to binary search on K such that $F(K) \geq N$.

Now the task is reduced to find $F(K)$ for a given K. This can be easily found with digit dp.

Let B be the binary representation of K and its length be len. Consider 1-based indexing in rest of the editorial.
Let $dp1[pos][tight]$ represent the count of numbers which are $\leq B[pos:len]$.
Here $B[pos:len]$ represents the substring of B starting from index pos upto end.
Here $tight$ is a boolean which represents if previous bits ie from index 1 to $pos-1$ are chosen exactly same as they are in B.

Now first let's calculate $dp1$.
Case 1: $tight = 1$
This means all previous bits have been chosen as it is, hence current bit can be selected $<= B[pos]$. Otherwise current number would be greater than K. Hence,

- If $B[pos] = 0$, we can select current bit as only 0 so dp transition would be as following:
  $dp1[pos][tight] = dp1[pos+1][tight]$.
- If $B[pos] = 1$, we have two choices for current bit as either 0 or 1. If we choose 0, $tight$ would become 0 as the number would no more remain same as B upto current index. And if we choose current digit as 1, $tight$ would remain 1 only so dp transition would be as following:
  $dp1[pos][tight] = dp1[pos+1][0] + dp1[pos+1][1]$.

Case 2: $tight = 0$
If $tight = 0$, then we can use either 0 or 1 for current bit as the number would always be $< K$ hence dp transition would be:
$dp1[pos][tight] = dp1[pos+1][tight] + dp1[pos+1][tight]$.

Calculating F(K) with DP

Now we will use this dp to calculate F. Let $dp2[pos][tight][parity]$ represent the sum of odd bits of numbers which started from an index with odd-even parity as $parity$ and are $\leq B[pos:len]$.
Here $B[pos:len]$ represents the substring of B starting from index pos upto end.
Here $tight$ is a boolean which represents if previous bits ie from index 1 to $pos-1$ are chosen exactly same as

they are in B.
Here if start index was odd then $parity = 1$, if start index was even then $parity = 0$. Initially we will say $parity = 2$ if start index is not known.

Now lets calculate $dp2$. Two cases here as well:
Case 1: $tight = 1$
Here if $B[pos] = 0$, current bit won't contribute anything to the sum as its 0. Sum value would be equal to the sum we get from next index. Hence dp transition can be written as following:

- $dp2[pos][tight][parity] = dp2[pos + 1][tight][parity]$

If $B[pos] = 1$

- We can either select 0 for current bit
  - In that case, $tight$ would become 0 and sum would be equal to the sum we get from next index ie $dp2[pos + 1][0][parity]$.
- Or we can also choose 1 for current bit
  - If $parity = 2$ then it would become $parity = pos\%2$ as pos would be start position because of picking current bit as 1.
  - Current bit would contribute only if the current pos is at odd position from start ie $(pos - parity)\%2 = 0$.
  - Also the contribution from current bit would be $dp1[pos + 1][tight]$. Because the count of numbers which can be formed $<= K$ after fixing current bit as 1 is $dp1[pos + 1][tight]$.
  - Contribution from next index ie $dp2[pos + 1][tight][pos\%2]$ would also be added as we are finding sum.
- To find the sum, we add the values we get from both choices.

Hence the dp transition can be written as following:

$$dp2[pos][tight][parity] = dp2[pos + 1][0][parity] + dp2[pos + 1][tight][pos\%2] + ((pos - parity)\%2 == 0)?dp1[pos + 1][tight] : 0.$$

Case 2: $tight = 0$
In this case, we always have 2 choices for current bit.

- If we select current bit as 0, the sum would be equal to $dp2[pos + 1][tight][parity]$.
- If we select current bit as 1
  - If $parity = 2$ then it would become $parity = pos\%2$ as pos would be start position because of picking current bit as 1.
  - Current bit would contribute only if the current pos is at odd position from start ie $(pos - parity)\%2 = 0$.
  - Also the contribution from current bit would be $dp1[pos + 1][tight]$. Because the count of numbers which can be formed $<= K$ after fixing current bit as 1 is $dp1[pos + 1][tight]$.
  - Contribution from next index ie $dp2[pos + 1][tight][pos\%2]$ would also be added as we are finding sum.
- To find the sum, we add the values we get from both choices.

Hence the dp transition can be written as following:

$$dp2[pos][tight][parity] = dp2[pos + 1][tight][parity] + dp2[pos + 1][tight][pos\%2] + ((pos - parity)\%2 == 0)?dp1[pos + 1][tight] : 0.$$

TIME COMPLEXITY:

$O(10 \cdot log^2 n)$ per test case

PROBLEM:

You are given a positive integer X which is at most 10^8.

Find three **distinct** non-negative integers A,B,CA,B,C that do not exceed 10^9 and satisfy the following equation:

$$(A|B)\&(B|C)\&(C|A) = X$$

Here, | denotes the **bitwise OR operator** and & denotes the **bitwise AND operator**.

It can be shown that a solution always exists for inputs satisfying the given constraints. If there are multiple solutions, you may print any of them.

EXPLANATION:

Every set bit of X must be present in (A|B), (B|C) and (A|C) which also boils down to the fact that every set bit of X must be present in at least 2 of A,B and C.
So, we can make a construction as A = X = B, C= 0. To make A!=B we can add a larger power of 2 to A(say 2^{27}), making sure it does not get larger than 10^9.

TIME COMPLEXITY:

O(1) for each test case.

PREREQUISITES:

Bitwise OR, Combinatorics

PROBLEM:

Let's define $Score$ of an array B of length M as $\sum_{i=1}^{M}\sum_{j=i}^{M} f(i,j)$, where $f(i,j) = B_i \mid B_{i+1} \mid \ldots \mid B_j$ (Here $\mid$ denotes the bitwise OR operation)

You are given an array A, consisting of N **distinct** elements.

Find the sum of $Score$ over all $N!$ possible permutations of A.

Since the answer can be large, output it modulo 998244353.

QUICK EXPLANATION:

- Contribution of i^{th} bit in the *Score* is independent of the contribution of j^{th} bit, for $i \neq j$.
- Sum of *Scores* over all permutations can be written as - sum of contribution of i^{th} bit (for $1 \leq i \leq 30$) over all permutations.
- Find total number of subarray (over all permutations) such that the i^{th} bit is not set in the OR of these subarrays. Using this, calculate number of subarrays num_i such that the i^{th} bit is set in the OR of these subarrays.
- Contribution of the i^{th} bit = $2^{i-1} \cdot num_i$, where $num_i = (N+1)! \cdot (\frac{N}{2} - \frac{K}{N-K+2})$ and K represents the number of numbers in which the i^{th} is not set.

EXPLANATION:

Score of an array is defined as the sum of the OR of all the possible subarrays. Now, an important observation regarding OR (which holds for other bitwise operators too) is that the result of the i^{th} bit is independent of the result of the j^{th} bit. This observation suggests us that if we can focus on a fixed bit, and can calculate the contribution of that bit in the *Score*, then we can solve our problem.

Let us focus on the i^{th} bit. Let num_i denotes the number of subarrays such that in the OR of these subarrays, the i^{th} bit is set. This is equivalent to saying that at least one element in each of these subarrays have the i^{th} bit set. Now, it is easier to calculate the number of subarrays in which none of the elements have the i^{th} bit set. Hence, we can write num_i as follows:

num_i = Total number of subarrays - number of subarrays in which none of the elements have the i^{th} bit set

Total number of subarrays

If we consider a single permutation, then the number of subarrays in that permutation is $\frac{(N)(N+1)}{2}$.
Total number of permutations = $N!$, where ! represents factorial.
Hence, total number of subarrays = $N! \cdot \frac{(N)(N+1)}{2}$

number of subarrays in which none of the elements have the i-th bit set

Let K represents the number of numbers in which the i^{th} bit is not set.
We will calculate the required number of subarrays by calculating the subarrays of length len ($1 \leq len \leq K$) for each len individually. Note that all the subarrays of length $len \geq K+1$ will have at least one element with the i^{th} bit set.

Now, the number of subarrays of length len satisfying the required property
$= \binom{K}{len} \cdot (len)! \cdot (N - len)! \cdot (N - len + 1)$

First term denotes the number of ways to choose len elements out of the K elements which have the i^{th} bit as not set. The second term denotes the number of ways to arrange these len elements and third term denotes the number of ways to arrange the remaining elements. The fourth term denotes the number of positions where this subarray of length len can be placed in the entire array.

Summing this expression for all the values of len

After simplifying, the above expression can be written as

$(K)! \cdot (N - K + 1)! \cdot \binom{N-len+1}{N-K+1}$

Summing over all values of len, we get

$\sum_{len=1}^{K} (K)! \cdot (N - K + 1)! \cdot \binom{N-len+1}{N-K+1}$

$= (K)! \cdot (N - K + 1)! \cdot \sum_{len=1}^{K} \binom{N-len+1}{N-K+1})$

The summation term can be represented as the sum of coefficients of x^{N-K+1} in $(1+x)^{N-len+1}$.
Coefficient of x^{N-K+1} : $(1+x)^{N-K+1} + (1+x)^{N-K+2} + \cdots + (1+x)^{N}$.

We can first simplify the Geometric Progression at the Right Hand Side, and say that the above value is equivalent to coefficient of x^{N-K+2} in $(1+x)^{N+1}$.

Substituting this value back in the original expression, we get the final value as $\frac{(N+1)! \cdot K}{N-K+2}$

Hence, $num_i = (N+1)! \cdot (\frac{N}{2} - \frac{K}{N-K+2})$

Contribution of the i^{th} bit = $2^{i-1} \cdot num_i$. So finally, we need to sum up this contribution over all values of i to get the final answer.

TIME COMPLEXITY:

We can pre-compute all the factorials beforehand : $O(N)$
Then for each test case, calculate K for each i - $O(N \cdot \log A_i)$, and use these values to get answer for each i.
Overall Time Complexity: $O(N \cdot \log A_i)$ for each test case.

PROBLEM:

You are given a string S consisting of only the characters P and N. In one move, you can flip a single P to N or vice versa. Find the minimum number of moves required to obtain a string that can be partitioned into an equal number of P and NP substrings.

EXPLANATION:

Suppose S can be partitioned into an equal number of P and NP substrings. Let's make a few observations:

- The last character of S cannot be N, because if this were the case, it cannot be part of either a P or an NP substring — but each character must be part of one such substring.
- In the same vein, NN cannot occur as a substring in S, because the first N cannot occur in either a P or an NP substring.
- Each N occurs in exactly one NP substring, and the number of these equals the number of P substrings. Thus, the number of occurrences of N in the string must be exactly $|S|/3$.

It turns out that these above conditions are also sufficient, i.e, if S is a string such its last character is P, no two occurrences of N are next to each other in S, and there are exactly $|S|/3$ occurrences of N in S, then there exists a way to split S into P and NP substrings such that there is an even number of each.

Proof

There are exactly $|S|/3$ N-s in the string, and since no two of them occur next to each other and the last character is not N, each of the N-s also has a P immediately after it. This gives us $|S|/3$ substrings of the form NP, using $2|S|/3$ characters.
The remaining characters are all P, and there are exactly $|S| - 2|S|/3 = |S|/3$ of them, which also gives us $|S|/3$ P substrings, and so we are done.

So, all we need to do is compute the number of moves to bring the string to this form.
First, let's deal with the condition on the last character - if it is N, we have no choice but to use one move to make it P.

Now, let's deal with the second condition.
Consider a substring of S whose characters are all N. Suppose the length of this substring is k.
We know that in the final string, we cannot have two N adjacent to each other. Thus, we can keep at most $\lceil \frac{k}{2} \rceil$ of these N-s — in other words, this substring needs at least $\lfloor \frac{k}{2} \rfloor$ moves to 'fix'.

Finally, let's deal with the third condition.
From the first two conditions, we have used some moves to make the string such that the last character is P, and no two N are adjacent.
Suppose the string currently has k occurrences of N.

- If $k = |S|/3$, no more moves are required.
- If $k > |S|/3$, simply pick any $k - |S|/3$ N-s and delete them. This satisfies the third condition while not breaking the first two

This leaves us with the case when $k < |S|/3$, where we need to turn some P-s into N while maintaining the first two conditions. Of course, this requires, at minimum, $|S|/3 - k$ moves.
It turns out that it is possible to do this using exactly $|S|/3 - k$ moves, with a fairly simple construction.

Construction

Split the string into $|S|/3$ blocks of three characters each.
There are $k < |S|/3$ N-s, so at least $|S|/3 - k$ of these blocks don't have any N at all.

Pick $|S|/3 - k$ blocks without an N, and turn the middle character of these blocks into N.
It's easy to see that all three constructions are now satisfied.

Every move we made was essentially forced by one of the three conditions, so the number of moves we made is optimal.

TIME COMPLEXITY:

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

PREREQUISITES:

DP with Bitmasking

PROBLEM:

It's Valentine's Month, and so Chef wants to gift a string to Chefina. He knows that Chefina likes **palindromic** strings. Chef has a string S consisting of digits from 0 to 9. He wants to convert this to a palindrome by performing zero or more operations, where an operation is defined as follows:

- Pick any digit and replace **all of its occurrences** with any other digit.

For example, if S=12123, in one move it can be turned into 14143 or 12128, but **not** 32123.

You are also given N integers $cost_1, cost_2, \ldots, cost_N$ where the cost of performing an operation that replaces X characters is $cost_X$.

Your task is to help Chef minimize the cost of converting his string to a palindrome.

QUIICK EXPLANATION:

Which numbers will convert to the same number at the end?

This can be found by using disjoint set unions and the i^{th} and the $(n+1-i)^{th}$ character will belong to the same group. So, some groups will be formed denoting that numbers of a particular group will finally convert to a single number.

How to calculate the answer of a formed group?

Our task is to convert all numbers of a particular group into a single number by doing multiple operations. We can use the bitmasking dp technique to calculate this.

The hidden catch

It might be possible that when two groups are transformed into a single number the cost is lesser. This arises due to the nature of the cost array.
To solve this issue, we can again use dp with bitmasking to choose the best answer.

EXPLANATION:

The numbers at s[i] and s[n+1-i] will finally convert to the same number and so will all their other occurrences. So, we can form groups for the numbers which will convert to the same number. This can be done using dsu. Since there are only 10 possible groups this can be done without any optimizations.

After the group formations, we want to calculate the answer for a particular group to convert them into a single integer. To do this we can use the dp with bitmasking technique. The final step of the conversion will involve two subgroups merging into one. We can iterate through all the subgroups of the group and find the new cost using the cost of the subgroups and the cost of this operation. The relation will look like
$dp[mask] = min(dp[mask], dp[submask] + dp[mask - sumask] + cost[number\ of\ elements\ in\ submask])$
We can pick the operation with the minimum cost. Iterating on all the submasks of all masks takes O(3^{10}) time.

But we are not done yet. Simply adding the cost of each group(to convert them into a single integer) won't e our answer. It may be optimal to convert two groups into a single integer because of the non-monotonic nature of the cost function.
Consider the string 1231 and the cost array as 9 1 2 4, it is optimal to convert 2 1's into 2 and then 3 2's into 3, rather than just converting 2 to 3.
So we again have to use the dynamic programming with bitmasking approach to update the dp.
In the previous step we have calculated the minimum cost to convert every possible mask(hence a group) into a single number. In this step we will update the dp values such that if a mask if formed of two or more groups(these are the groups formed after the dsu step) its cost can be directly the addition of the cost of the

subgroups.
$dp[mask] = min(dp[mask], dp[submask] + dp[mask - submask])$, provided that the submask is a collection of 1 or more complete groups(these are the groups formed after the dsu step). If we proceed in this manner and update the whole dp array, our final answer will be equal to ans[1023], since 1023 is the mask of the all the numbers together in a group.

TIME COMPLEXITY:

O($3^{10} + n$) for each test case, as the time to iterate through all submasks for all possible masks is 3^{10}, and O(n) computations to count the number of occurrences of a number and form groups among them.

PREREQUISITES:

Strings, Palindromes, Graphs, Floyd-Warshal

PROBLEM:

You are given a string S consisting of N digits from 0 to 9. You want to obtain a palindrome from S using some operations.

There are M different types of operations you can make on S. Each operation is described by three integers X, Y and W. In this operation, if there is a digit equal to X in S, you can replace X with Y using W coins.

You can do the operations in any order. One type of operation can be done multiple times. You can use at most K coins total in all operations. Among all the palindromes that can be obtained from S, output the lexicographically maximum palindrome.

EXPLANATION:

Let's say S is the given string and P is the palindromic string that we are looking for. Assume that the i_{th} digit of both the strings are different. Do we care about the type of operations or number of operations that we took to convert S_i into P_i?

No, we don't. The only thing that matters for us is the number of coins that were required to convert S_i into P_i. Hence, we now need to find out the minimum number of coins that will be required to convert S_i digit into P_i. How can we do that?

Let us try to build a directed graph from the given operations:

What are the nodes of this graph?

Every digit from 0 to 9 represents a node of the graph.

What are the edges of this graph?

We are given operations as X, Y and W, hence we will have a directed edge from X to Y with a cost of W.

Now, we can use Floyd-Warshal Algorithm in this graph to find out the minimum coins that are needed to covert every digit into every other (i.e. we obtain the cost for every possible conversion $\{(x \rightarrow y) \forall (x \in [0,9], y \in [0,9])\}$, where the x represents the initial digit, and y represents the digit it is converted to).

Now, let's move towards the other part of the problem and introduce K in our approach. Due to the limit of K we need to find first whether there exists any palindromic string that can be formed by no more than K coins. For that, we can try to find out the minimum cost palindromic string.

Recall that for the string to be a palindrome, it's $(i)_{th}$ and $(n-i-1)_{th}$ digit should be the same. As we are looking to find out the minimum cost palindromic string, then we will try to find the minimum cost that will be required to make $(i)_{th}$ and $(n-i-1)_{th}$ digit same. To do so we can try to convert $(i)_{th}$ and $(n-i-1)_{th}$ digit to every possible digit and whichever gives us the minimum cost we will add that cost to our running cost.

Finally, we would be able to find the minimum cost palindromic string. If the minimum cost is greater than K then it won't be possible to get any palindromic string and hence we can simply output -1.

Otherwise, we will try to find the lexicographically largest palindromic string possible with the given cost of K. We can go greedy to find out the same.

Suppose we are at the i_{th} digit, to make the palindromic string lexicographically largest, we will try to make this i_{th} digit as large as possible. Remember $(i)_{th}$ and $(n-i-1)_{th}$ digit should be same, we can covert $(i)_{th}$ and

$(n-i-1)_{th}$ digit to digit d if and only if the coins left out after converting are enough to make substring $[i+1, n-(i+1)-1]$ palindromic.

Finally, we get the lexicographically largest palindromic string by using at most K coins. Well, don't forget to make the middle character as large as possible if the length is odd and you still have some coins left in your pocket.

Here is how we can efficiently find the minimum cost to make substring $[i, n-i-1]$ palindromic:
Recall that we have already calculated the minimum cost to make $(i)_{th}$ and $(n-i-1)_{th}$ digit same. Let us store that in an array and find the suffix sum of this array, then our $suff[i]$ denotes the minimum cost to make substring $[i, n-i-1]$ palindromic.

TIME COMPLEXITY:

$O(N.d)$ per test case where d is the number of digits.

PREREQUISITES

Observation, Greedy

PROBLEM

We call a number x good if its binary representation is palindromic. For e.g. 107 is good because its binary representation is 1101011 which is palindromic while 10 is not good because its binary representation is 1010 which is not palindromic.

You are given a number n. Express it as a sum of at most 12 good numbers. If there exist multiple answers, print any one of them.

QUICK EXPLANATION

- Build a list of integers in the range $[1, 1000]$ such that their binary representation is of the form $100\ldots001$ and store it in decreasing order.
- For a given n, we can repeatedly pick the largest element in the list and subtract it from n.
- Since, at each subtraction, the number is halved, the process takes only $O(log_2(N))$ operations.

EXPLANATION

In this solution, we consider only the numbers, whose binary representation are of the form $100\ldots001$ and are in the range $[1, 1000]$. We can see that the binary representation would be a palindrome. The list comes out to be $[1, 3, 5, 9, 17, 33, 65, 129, 257, 513]$.

Let's try to answer a query, only using these numbers greedily. For current n, pick the largest number in this list $\leq n$, and add it to the answer list and subtract it from n.

The crucial observation here is that n is reduced by at least $n/2$. Ignoring element 1, we can see that consecutive elements are of the form x and $2 * x - 1$. If x is the largest element $\leq n$, then $x \leq n < 2 * x - 1$. Dividing all terms by 2, we have $x/2 \leq n/2 < x - 0.5$. Focusing on $n/2 < x - 0.5$, we can see that $n/2 \leq x$, hence n is reduced by half every time.

This way, halving at each step, the number of operations does not exceed $log_2(N)$, which is less than 12.

Note: Solutions used by testers used the DP approach, which essentially boils down to the above thing. They used all good numbers, unlike setter.

TIME COMPLEXITY

The time complexity is $O(M * log(M) + T * log(N))$ where $M = 1000$

PREREQUISITES

Sieve of Eratosthenes

PROBLEM

Chef has the N numbers $1, 2, 3, \ldots, N$. He wants to give exactly K of these numbers to his friend and keep the rest with him.

He can choose any K numbers such that the GCD of any number from Chef's set and any number from his friend's set is equal to 1.

Formally, suppose Chef gives the set of numbers A to his friend and keeps B with himself (where $|A| = K$ and $|B| = N - K$). Then A and B must satisfy

$$\gcd(a, b) = 1 \quad \forall a \in A, b \in B$$

Chef needs your help in choosing these K numbers. Please find any valid set of K numbers that will satisfy the condition, or tell him that no such set exists.

QUICK EXPLANATION

- Consider a set A containing an integer 1 and all prime numbers strictly greater than $N/2$. Let's assume the size of this set is C.
- The rest of the elements must be in one set B only. So if both K and $N - K$ are $< N - C$, there's no possible way to divide.
- Otherwise, we can keep adding elements from set A to B until one of them is of size K. We can print the set with size K.

EXPLANATION

Graph formulation

There shouldn't exist any pair (a, b) such that $gcd(a, b) > 1, a \in A, b \in B$. Let's consider a graph where for every pair (a, b), an edge is present if and only if $gcd(a, b) > 1$.

We cannot put any two vertices from the same connected component to the different sets.

For example, for $N = 6$, we have edges $(2, 4), (2, 6), (4, 6), (3, 6)$. This way, we get connected components $[(1), (2, 3, 4, 6), (5)]$.

Assuming $K = 4$, we can print `2 3 4 6` as a valid set. for $K = 2$, `1 5` works.

So, assuming we can get the sizes of connected components, can we find to select some components having total node count equal to K. This seems like a knapsack problem. So we need to think more.

Number theory

Let's focus on prime numbers since the gcd function works on primes independently of other primes.

Consider all primes p such that $p * 2 \leq N$. Then we have edges $(2, 2 * p)$ and $(p, 2 * p)$. Hence, 2 and p must be in same component for all primes $\leq N/2$. Let's add this to a set called S.

All non-prime integers shall also lie in the same set as their prime factors. So any integer > 1 which is not a prime shall also be added to this S.

We can claim that the elements present in S shall all be in either A or B.

For example, for $N = 13$, the primes less than 6.5 are $2, 3, 5$, so the set S formed is $[2, 3, 4, 5, 6, 8, 9, 10, 12]$.

Elements not present in this set are 1 and all primes greater than $N/2$. Let's say there are C such elements. for $N = 13$, we have $[1, 7, 11, 13]$. $C = 4$ here.

In the graph, the set S represents a single component, and the rest all appear as connected components of size 1 each.

Implementation

So, if we can add elements from $[1, 7, 11, 13]$ to make a set of size K or size $N - K$, then it is possible to find such A and B.

It is only when we have either $K \leq C$ or $K \geq N - C$ that we can form set A and B.

Considering $N = 13$ and $K = 11$, set $A = [2, 3, 4, 5, 6, 8, 9, 10, 12, 1, 7]$ is a valid answer. For $K = 3$, $A = [1, 11, 13]$ is a valid answer.

The list of primes can be computed using the sieve of Eratosthenes.

TIME COMPLEXITY

The time complexity is $O(N * log(log(N)))$ per test case due to sieve.

PROBLEM:

Alice and Bob are playing a game. In this game, there are N piles of stones, say $A_1,A_2,…A_N$In his move, the player can select any of the piles, discard the rest of the $(N-1)$ piles, and partition the selected pile into N piles containing at least 1 stone. The player who cannot make a move loses. Determine the winner of the game if Alice moves first

Quick Explanation

Winning state can be define here as:

$$x \ mod \ n(n-1) = \{0 \ or >= n\}$$

All other states are losing state. So just iterate through the array and check if there is a winning state then Alice would win the game else Bob would win.

Explanation

The first step for a player would be to check if there exists a pile choosing which he can win the game, i.e a pile which can be split into n piles $\{p1, p2, p3....pn\}$ such that $p_i < n \ \forall \ i$. Thus the question reduces to choosing a pile that satisfies the above condition. We will perform this check for each pile so now for a pile having x stones we want to check whether it satisfies our condition or not. This can have three cases:

- Case 1: $1 \leq x < n$: Here we can clearly see that the player cannot split the given pile into piles since x is less than n.
- Case 2: $n \leq x \leq n(n-1)$: Here it is easy to see that the player can divide the pile into n piles each having less than n stones.
- Case 3: $n(n-1)+1 \leq x < n(n-1)+n$: Here no matter how he divides the piles there will always be a pile having at least n stones. so he will lose.
- Case 4: $n(n-1)+n \leq x \leq 2n(n-1)$: We can see that player 1 can lead to all losing states by putting 1 stones in $(n-1)$ piles and $n(n-1)+1$ stones in the n_{th} pile.

Thus using similar pattern analysis we can reach our initial claim in quick explanations.

A little more Mathematical proof:

$$Claim : x \ mod \ n(n-1) = \begin{cases} 1 \leq r \leq n(n-1).....Losing \ State \\ n \leq r \leq n(n-1).....Winning \ State \end{cases}$$

Proof by Contradiction

Assuming $k = qn(n-1)+r$ is the smallest winning state where $1 \leq r \leq (n-1)$. Here k can be represented as a sum of n losing states. Now all smaller losing states are such that $1 \leq rem \leq (n-1)$. This implies:

$$\sum_{i=1}^{i=n} 1 \leq \sum_{i=1}^{i=n} rem_i \leq \sum_{i=1}^{i=n} (n-1)$$

$$=> n \leq r \leq n(n-1).....Contradiction$$

Proving for the 2nd part. Assuming k is a losing state and $n \leq r \leq n(n-1)$. This implies that k cannot be represented as sum of n losing states. Now observe that r can be constructed as follows:

Assume $k' = qn(n-1) + rem'$ as 1st division. Divide remaining $(r - rem')$ among remaining $(n-1)$ part. This reduces to the problem that sum such that $n \leq sum \leq n(n-1)$ can be constructed using n items, each of which holds $1 \leq item \leq (n-1)$ which again makes a contradiction.

TIME COMPLEXITY:

Here we simply need to perform a mod operation on each elements of the array. The mod operation would take $O(1)$ time and iterating through the array would take $O(N)$ time. Thus in total the time complexity would be $O(N)$.

PREREQUISITES:

Bitwise XOR

PROBLEM:

You are given an integer N. Construct a permutation A of length N which is **attractive**.

A permutation is called **attractive** if the bitwise XOR of all absolute differences of adjacent pairs of elements is equal to 0.

Formally, a permutation $A = [A_1, A_2, \ldots, A_N]$ of length N is said to be attractive if:

$$|A_1 - A_2| \oplus |A_2 - A_3| \oplus \ldots \oplus |A_{N-1} - A_N| = 0$$

where $\oplus$ denotes the bitwise XOR operation.

Output **any** attractive permutation of length N. If no attractive permutation exists, print -1 instead.

Note: A permutation of length N is an array $A = [A_1, A_2, \ldots, A_N]$ such that every integer from 1 to N occurs exactly once in A. For example, $[1, 2, 3]$ and $[2, 3, 1]$ are permutations of length 3, but $[1, 2, 1]$, $[4, 1, 2]$, and $[2, 3, 1, 4]$ are not.

QUICK EXPLANATION:

If N is odd

In this case, there are total even number of absolute difference terms. If these terms are all equal, their overall XOR will become 0.
Hence, we can have all absolute difference terms to be equal to 1. In simpler words, the permutation $\{1, 2, \ldots N\}$ is a valid permutation.

If N is even

In this case, there are total odd number of absolute difference terms.
We cannot make XOR of 1 term to be 0 (as each term is greater than 0). Let us try to make XOR of first 3 terms to be 0, and follow the above strategy for rest of the terms.

One of the valid permutation is: $\{2, 3, 1, 4, 5, 6 \ldots N\}$.

Corner Case

$N = 2$ is a corner case. Since we have only one absolute difference term, and it is non-zero, we cannot have the overall XOR as 0. The answer would be -1 in this case.

EXPLANATION:

There can be many possible constructions. This editorial explains one of the strategies to make a valid permutation.

Let us divide our solution in two parts depending on the parity of N. We will first see a valid construction when N is odd, and then we will use this solution to construct a valid permutation for even values of N. Finally, we will take a look at a corner case.

When N is odd

In this case, we will have even number of absolute difference terms. Now, whenever we deal with overall XOR to be equal to 0, one of the properties of XOR that we should keep in mind is: $a \bigoplus a = 0$. Let us extend this property for our case. If we take even number of $a's$ and take their XOR, the result will be equal to 0.

So, if all the absolute terms are equal, we will have a valid permutation. One of the permutation that satisfies this property is $\{1, 2, \ldots, N\}$

When N is even

In this case, we will have odd number of absolute difference terms. Also, we already have a solution when we have even number of absolute difference terms. Let us think in the direction of breaking these odd number of terms in two parts - one having odd number of terms such that its XOR is 0, and other having even number of terms. Because we can't have XOR of 1 term as 0 (because each term is greater than 0) , let us focus on 3 terms.

Let the permutation be $\{P_1, P_2, \ldots P_N\}$. The above parts corresponds to terms from the following two sequences - $\{P_1, P_2, P_3, P_4\}$ and $\{P_4, P_5, \ldots P_N\}$.

We have already seen that if $\{P_4, P_5, \ldots P_N\} = \{4, 5, \ldots, N\}$, then we have the XOR of absolute difference terms as 0. Also, this fixes the value of P_4 as 4, and leaves $1, 2$ and 3 for P_1, P_2 and P_3.

A little trial and error on paper shows that permutations like $\{1, 3, 2, 4\}$ or $\{2, 3, 1, 4\}$ have their XOR of absolute difference terms as 0.

Hence, one of the valid permutations is $\{2, 3, 1, 4, 5, 6 \ldots N\}$.

An Important corner case

Note that if $N = 2$, we have only one absolute difference term, and it is non-zero. So we cannot have the overall XOR as 0. The answer would be -1 in this case.

TIME COMPLEXITY:

$O(N)$ for each test case.

PREREQUISITES:

Implementation, Running Sum

PROBLEM:

Suppose we have a $2D$ grid A of infinite width and height of N units. Each row of the grid can be represented in the form of $111\ldots$(x times)$00\ldots\infty$ where $x > 0$, implying the row starts with consecutive $1_s'$ followed by all $0_s'$.

Now you do the following in each step.

```
Choose an unmarked cell having 1. Let it be (i, j).
while((i,j) lies in the grid boundaries &amp;amp;&amp;amp; A[i][j] == 1){
    mark[i][j] = true; i--; j--;
}
```

How many minimum number of steps are required to cover all the cells containing 1.

Note: The number of consecutive $1_s'$ at each height is given by the array W where W_i represents the same at i_{th} height.

QUICK EXPLANATION:

We will apply the step on all the 1's in the first row. Then we move down the rows and check how many set bits are to the bottom left of set bits of the row above it, and apply no extra steps to make these 0. For the 1's in a row that have no 1 to the top right of themselves, extra step must be applied. Maintain this running sum of needed number of steps to make all set bits unset from the top to the current row of the matrix in order to obtain the final answer.

EXPLANATION:

Claim: If the number of 1's (say L) in a row are less than the number of 1's (say U) in the row above it, then the row above can engulf all the 1's of the row below it without using any extra step.

Proof

Assuming U and L as above, we are working on the implications of the relation $L < U$.

Consider the topmost row consisting of $x + 1$ number of 1's: $1_0 1_1 1_2 ... 1_x 000...$
For the first 1 in the row i.e. at $A[0][y]$, there is no element $A[i-1][j-1] = A[-1][y]$ (to its bottom left as per the diagram in the problem statement itself) that can be engulfed in a single step along with $A[0][0]$. The same is also true for every first element in every row no matter what height it recides at (i.e. value of y is of no consequence). Thus the first element of none of the rows can engulf any of the elements in the lower row in a single step but the next x number of 1's can. Thus a row with $x + 1$ number of 1's can engulf x number of 1's from the row below it.

The second 1 of the upper row at $A[1][y]$ can engulf the 1 at $A[0][y-1]$.
The third 1 of the upper row at $A[2][y]$ can engulf the 1 at $A[1][y-1]$.
Each 1 can engulf the 1 diagonally to the bottom left of itself, thus the last 1 in a row $A[x][y]$ can engulf $A[x-1][y-1]$. If the 1 engulfed by any of the 1's of the upper row was the last 1 of the lower row, then no extra step is needed to be performed on this lower row as all its 1's became 0's when we selected each of the 1's of the upper row and went diagonally bottom left in each step.

As the topmost row has no other row above it, we apply the step on all of its 1's as they can't depend on a higher row element to engulf them. Then we move to the next row and check whether the 1's can be engulfed or

not. The ones that can’t be engulfed need to become the starting points of newer steps and thus will be able to engulf diagonally bottom left positioned 1's without extra steps and so on.

Before proceeding to a lower row, the number of steps needed to flip all the 1's above that row have been added to the answer. Thus on reaching such a row all whose 1's can be engulfed by the row above it, no extra step would be needed to flip all the 1's as they are reachable from the indices to the top right of themselves containing 1.

If this engulfing of the 1's of a lower row is not possible by the higher row, i.e. the number of 1's in the row below is greater than or equal to the number of 1's in the row above, then the 1's starting from U_{th} to the end of the lower row can not be engulfed. An extra $L-(U-1)$ steps would be needed so that these ungulfable 1's can be flipped and the matrix from top row to the current row can be all unset bits.

We keep maintaining this running sum in a variable starting from the first row until the last row and obtain the final answer.

TIME COMPLEXITY:

$O(N)$

PROBLEM:

Given an array A of size N.
$Operation$: You can select two indexes i and j ($i < j$) and swap A_i and A_j.
$Alternating\ Sum = S = |A_1| - |A_2| + |A_3| - |A_4| + \ldots (-1)^{N-1} \cdot |A_N|$
With **atmost one** operation find the maximum $Alternating\ Sum$.

EXPLANATION:

Since,
$S = |A_1| - |A_2| + |A_3| - |A_4| + \ldots (-1)^{N-1} \cdot |A_N|$

It can be observed that using an operation is useful if and only if i and j are of different parity.
Let,
$S_1 = \sum_{i\ is\ odd} A_i$
$S_2 = \sum_{i\ is\ even} A_i$
Then,
$S = S_1 - S_2$

Let's suppose we swap A_i and A_j, where i is odd and j is even and new alternating sum will become S'. Then,
$S' = (S_1 - |A_i| + |A_j|) - (S_2 + |A_i| - |A_j|) = S + 2 \cdot (|A_j| - |A_i|)$
So, to maximize S', we need to maximize $|A_j|$ and minimize $|A_i|$.
Let,
$|A_i|_{min}$ be minimum value of $|A_i|$ where i is odd.
$|A_j|_{max}$ be maximum value of $|A_j|$ where j is even.
then **maximum** $Alternating\ Sum = \max(S, S + 2 \cdot (|A_j|_{max} - |A_i|_{min}))$

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

Sneil is given N points on a coordinate graph. From these points (X_i, Y_i) he can draw horizontal and vertical lines. If he finds the number of lines that can be drawn he will get to eat Sneilium.
Since Sneil is too slow he asks you to help him with this task.

EXPLANATION:

After we have drawn all the lines, we have a grid with some lines parallel to the X axis and some lines parallel to the Y axis. The total number of lines is just the **sum** of the lines parallel to the X axis and the lines parallel to the Y axis.

The only catch is to **handle duplicates**. For that, we can use set data structure. We club all the X coordinates and all the Y coordinates into separate sets.

Finally, the total number of lines is just the sum of the sizes of both these sets.

TIME COMPLEXITY:

Ordered sets have a complexity of $O(logN)$ per insertion. This leads to a time complexity of $O(NlogN)$ per testcase. We can use unordered sets to reduce this to $O(N)$ per test case.

PROBLEM

Given a circular array A of length N with non-negative integers and an integer K, the following process is repeated every second

- For each i such that $A_i > 0$, elements adjacent to i-th element get incremented by 1.

Find the sum of the array A after K seconds.

QUICK EXPLANATION

- Once an element becomes non-zero, it stays non-zero.
- Each non-zero element contributes 2 to the sum every second. If A_p becomes non-zero at time T, then it contributes $2 * max(0, K - T)$ to the overall sum.
- For each element, we can calculate the first time it becomes non-zero by finding the nearest non-zero values in the given circular array.
- Only edge case is when all elements are 0. The sum remains 0 in this case.

EXPLANATION

Instead of trying to maintain the actual array, let's only keep relevant information. We only care about the sum of the array after K seconds, so let's try computing that directly. The sum of the final array would be the sum of the initial array plus the increments from operations.

In each operation, for each non-zero A_i, one is added to adjacent elements on both sides, which increases the sum of the array by 2. This happens for each non-zero value in the array.

So, if before an operation, there are x non-zero values in an array, the sum is increased by $2 * x$ by that operation.

Also, we can see that once an element becomes non-zero, it only increases, and thus, keeps contributing to the overall sum.

Minimum time an element becomes non-zero

Now, the problem reduces to computing T_i, where T_i is the minimum time when A_i becomes non-zero. For non-zero elements in initial array, $T_i = 0$.

Let's assume A_p is the nearest non-zero value for A_i. Then it takes $|p - i|$ seconds, as, after each second, the element adjacent to the nearest non-zero element becomes positive, reducing $|p - i|$ by 1.

Hence, for each element, find the position of the nearest non-zero element in both directions, and take the minimum distance to both elements.

It can be easily done via ordered sets in $O(N * log(N))$.

But we can use DP as well, using recurrence $T_{i+1} = min(T_{i+1}, T_i + 1)$ (looping i from 1 to N twice to handle circular array) and $T_{i-1} = min(T_{i-1}, T_i + 1)$ (looping i from N to 1 twice to handle circular array).

Hence, this way we are able to compute T_i, so we can compute sum of final array as $\sum_{i=1}^{N} A_i + 2 * max(0, K - T_i)$

Bonus

Solve this problem for non-circular array A.

TIME COMPLEXITY

The time complexity is $O(N)$ or $O(N * log(N))$ depending upon implementation

PREREQUISITES:

Bitwise Operations, Dynamic Programming

PROBLEM:

Chef has an array A such that $A_i = i$.

A *transform* on the array is performed by replacing each element of the array with the **bitwise XOR** of the prefix till that element. For example, if B denotes the array after performing *transform* on array A, then, $B_i = A_1 \oplus A_2 \oplus \ldots \oplus A_i$. Similarly, if C denotes the array after performing two *transforms* on array A, then, $C_i = B_1 \oplus B_2 \oplus \ldots \oplus B_i$.

Let f^K be the array obtained by performing K *transforms* on the array A. Thus, f^K_X denotes the X^{th} element of the K^{th} transform of the array A.

Formally,

- $f^0_i = A_i$
- $f^1_i = A_1 \oplus A_2 \oplus \ldots \oplus A_i$
- $f^2_i = f^1_1 \oplus f^1_2 \oplus \ldots \oplus f^1_i$
- $f^k_i = f^{k-1}_1 \oplus f^{k-1}_2 \oplus \ldots \oplus f^{k-1}_i$. Here, $\oplus$ denotes the bitwise XOR operation.

Given K and X, determine the value f^K_X.

EXPLANATION:

Basically in this problem, we need to calculate $f_k[i]$. By definition

$$f_k[i] = f_{k-1}[0] \oplus f_{k-1}[1] \oplus ....f_{k-1}[i]$$

Using simple combinatorics and induction, we can prove that $f_k[i]$ is just XOR of all $1 \leq j \leq i$ such that $\binom{k+i-j}{k}$ is odd.
This is a well known fact as a corollary of Lucas Theorem that $\binom{k+i-j}{k}$ is odd if and only if k is a submask of $(k + i - j)$.

Now we need to find XOR of all j such that $1 \leq j \leq i$ and $(k + i - j)$ is a supermask of k.

We can do it using $dp[bit][carry][0/1]$ which defines the XOR of all j if we processed the first bit bits, $carry$ is a carry bit for $(k + i - j)$ that we have after processing bit bits, and the last parameter is 0 if $j \leq i$ or 1 otherwise.

Now we go through bits, if the current bit of k is 0, we can try to make a bit of $(k + i - j)$ equals to 0 or 1, and if the current bit of k is 1, then the bit of $(k + i - j)$ should also be 1.

TIME COMPLEXITY:

$O(logK)$, for each test case

PROBLEM:

You are given two arrays A and B, each of size N. You can perform the following types of operations on array A.

- Type 1: Select any prefix of A and increment all its elements by 1.
- Type 2: Select any suffix of A and increment all its elements by 1.

Your task is to transform the array A into array B using the **minimum** number of operations. If it is impossible to do so, output -1.

For an array A having N elements:

- A **prefix** of the array A is a subarray starting at index 1.
- A **suffix** of the array A is a subarray ending at index N.

EXPLANATION:

Let us first try to solve the problem when the operations of only Type 1 are allowed.
Let us define a difference array D such that $D_i = B_i - A_i$ for $1 \leq i \leq N$.

Consider an index i such that $D_{i+1} > D_i$. Because we will have to add D_{i+1} to A_{i+1}, we will also end up adding D_{i+1} to A_i, as the operations of Type-1 applies on a prefix. This will make $A_i^{'} = A_i + D_{i+1} > A_i + D_i = B_i$. Hence, we end up getting $A_i^{'} > B_i$. This suggests that if $D_{i+1} > D_i$ for any valid i, then we cannot convert A into B using only operations of Type 1. Otherwise, we can convert A into B using D_1 operations by applying $O_i = D_i - \sum_{j=i+1}^{N} D_j$ operations on prefix of length i.

Now let us come back to our original problem. Let us again define the difference array D such that $D_i = B_i - A_i$ for $1 \leq i \leq N$. First of all, we will apply the operations of Type 2 only when absolutely necessary.

Consider an index i. If $D_i < D_{i+1}$ we have seen that this cannot be solved using operations of Type 1 only. Hence we must apply operations of Type 2. To be more specific, we need to apply at least $D_{i+1} - D_i$ operations of Type 2 on the suffix $A[i+1 \ldots N]$ so that the condition $D_i \geq D_{i+1}$ will hold. Let us apply these many operations of Type 2. We will update D_j for all $i < j \leq N$, and will continue the above process. If at any index, we have $D_i < 0$, then we cannot convert A into B (as we have only applied the minimum necessarily required operations till now). Otherwise, we will finally get an updated difference array, and we now need to apply D_1 additional operations of Type 1.

Implementation detail: To update D_j for all $i < j \leq N$, we can just maintain a running sum variable which will store the total number of Type 2 operations applied till that point. The updated D_i will be $D_i + sum$.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

Arun has an integer N. His friend Sucksum likes the number 1, so Arun wants to reduce N to 1.

To do so, he can perform the following move several times (possibly, zero):

- Pick two integers X and Y such that $X+Y$ is even and X^Y is a divisor of N. Then, replace N by $\frac{N}{X^Y}$

Note that at each step, X^Y only needs to be a divisor of the *current* value of N, and not necessarily a divisor of the integer Arun initially started with.

Find the **minimum** number of moves Arun needs to reduce N to 1. If it is not possible to reduce N to 1 using the given operation, print -1 instead.

EXPLANATION:

Let us tackle this problem case-wise for N, but before that we define another variable $count$ which stores the count of power of 2 for N.

- $N = 1$: Answer would be 0, since we already are at 1.
- $count = 0$. Answer would be 1 since then N would be odd so we can take $X = N$ and $Y = 1$. Then $X + Y$ would be even and X^Y would be N, that is a divisor of N which would reduce it to 1.
- $count\ is\ odd$: We cannot reach 1. Taking 2 here, we would observe that it cannot be reduced to 1. Similarly for any other number, we would at best reduce it to 2 and then get stuck.
- $N\ is\ a\ perfect\ square$: Here we take $X = \sqrt{N}$ and $Y = 2$. Since X would also have a power of 2 so it would be even, thus $X + Y$ would be even and X^Y would be N, that is a divisor of N which would reduce it to 1 so answer would be 1
- $Otherwise$: We are talking about cases where N is not a perfect square and $count$ is even. Here we would first take $X = 2^{count/2}$ and $Y = 2$. This would reduce N to $N^{'}$ that does not contain any power of 2. This would then be the same case as above so we would require one more move. Overall answer would be 2.

TIME COMPLEXITY:

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

PREREQUISITES:

Strings

PROBLEM:

A string consisting only of parentheses (and) is called a parentheses string. The balanced parentheses strings are defined recursively as follows:

- An empty string is a balanced parentheses string.
- If s is a balanced parentheses string, then (s) is also a balanced parentheses string.
- If s and t are balanced parentheses strings, then st (concatenation of s and t) is also a balanced parentheses string.

For a parentheses string, an operation on this string is defined as follows:

- Choose any **substring** of this string which is balanced.
- Remove this substring and concatenate the remaining parts.

Given a parentheses string S, find the **minimum** length of the string one can get by applying the above operation any number of times. Also find the **minimum** number of operations required to achieve this minimum length.

EXPLANATION:

The first observation that we can make is that on removing a balanced parenthesis from the string, it won't affect other balanced parenthesis in the remaining string. So what we can do is remove all the balanced parenthesis from the string and then on the final string left, we can calculate the number of operations by counting the number of missing segments in it.

In order to implement it we can replace the string by a vector v, where

$$v[i] = \begin{cases} (i+1), & if\ s[i] = (\\ -(i+1), & if\ s[i] =) \end{cases}$$

Now we can take another vector, say res and loop through the vector v and for any index i, which has a closing bracket, i.e $v[i] < 0$, if we can find a corresponding opening bracket at index j, that makes string $s[j...i]$, balanced, i.e we find res not empty and last element of res to be positive, we simply remove the last element of res, implying that substring $s[j....i]$ is removed, otherwise we simply add $v[i]$ to res

Thus in the end res would contain the indexes of the final string after removing all the possible balanced parenthesis from the original string.
The size of res would be the first part of the answer, i.e the length of the final string. As for the number of operations, we have the indexes of the characters of the final string, so we can just count the number of missing segments to get the second part of the answer.

TIME COMPLEXITY:

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

PREREQUISITES:

Greedy

PROBLEM:

You are given an array A of N distinct integers. You are also given an array B of $N-1$ distinct integers.
Determine the smallest possible **positive integer** X such that $B_i - X$ exists in A, for all valid i.
It is guaranteed that X exists.

EXPLANATION:

Sort A and B. Then A_1 is the smallest element of A, and A_N the largest. Similarly with B too.

Claim: X is either $B_1 - A_1$ or $B_1 - A_2$.

Proof

All that remains now is to set X to each of the two possible values and validate if $B_i - X$ exists in A, for all i.
This can be done efficiently using hash tables.
If both values are possible, select the **smallest positive** value of the two.

TIME COMPLEXITY:

Sorting A and B takes $O(N \log N)$ each. Checking if $B_i - X$ exists in array A, for all i, takes $O(N)$ using hash tables.
The total time complexity per test case is thus:

$$O(2 * (N \log N) + N + N) \approx O(N \log N)$$

PREREQUISITES:

DFS, BFS, Connected Components

PROBLEM:

Given a binary grid. 1 shows land and 0 shows water.
In a turn, a player can capture an uncaptured cell that has land, and keep capturing neighboring cells sharing a side if they are also land. The game continues till no uncaptured cell has land and each player wants to capture as big size of land as possible in total when the game finishes. Find the maximum size of land that Chef can capture (in number of cells) if he plays second, and both players play optimally.

QUICK EXPLANATION:

Run BFS or DFS and find sizes of all connected components with all ones. Sort the sizes array in decreasing order. Finally find the sum of alternate elements starting from index 1.

EXPLANATION:

In each turn, one player can start capturing a cell with value 1 and he will keep capturing until he keeps finding 1. So basically, the player captures one whole connected component in one turn.

As both players play optimally, first player will always try to get bigger components, similarly the second player. Hence largest component will be captured by first player, second largest by second player, third largest by first player again, fourth largest by second player again and so on.

Hence we can find the sizes of all connected components with all ones. And sort the size array in decreasing order. Then find sum of alternate elements starting with index 1. (Assuming 0-based indexing). This will give us the total sum, second player can achieve.

TIME COMPLEXITY:

$O(n \cdot m \cdot \log{(n \cdot m)})$ per test case

PREREQUISITES:

Graphs

PROBLEM:

The country of Chefland is a group of N cities having M directed roads between its cities. It takes 1 second to travel on any road.

Chef has decided to travel through this country. His travel plan goes as follows -

- He will randomly choose a starting city.
- While there are outgoing roads in his current city, he will randomly choose one of them and travel it. He can choose roads he has already traveled through and visit cities multiple times.
- He will stop if there are no outgoing roads in the current city or N seconds have passed.

Chef will be disappointed if after traveling he does not end up in the capital city.

The ministers have not yet decided upon the capital city. They would like to choose it in such a way that Chef is not disappointed by his trip. To make their decision a bit easier, they can build some new roads between cities.

Find the **minimum** number of roads that need to be built such that the ministers can choose a capital city that doesn't disappoint Chef no matter where his trip starts or how it progresses.

It is possible that the current road system is such that no matter which roads are added, a capital that doesn't disappoint Chef cannot be chosen. In this case, print -1 instead.

QUICK EXPLANATION:

$Observation\ 1$: The candidates that can be a capital city cannot have any outgoing edges.
$Observation\ 2$: If the graph contains any cycle, then there cannot be a capital city.

$Claim$ For any k candidates for capital cities, the number of edges that needs to be added to make a unique capital is $(k-1)$

Thus we would first check for cycles in the graph and if no cycle is present in the graph, then we would count the number of vertices having no outgoing edges and answer would be one less than that.

TIME COMPLEXITY:

$O(V+E)$ for each test case.

PREREQUISITES:

Z-function, RMQ data structures

PROBLEM:

Given two arrays A and B of length N each.

You can do the following operation on the array B **exactly once**:

- Chose two integers R $(1 \leq R \leq N)$ and $K(K \geq 0)$ and apply K right-rotations to the subarray $[1, R]$. Applying a right-rotation to the array $C = [C_1, C_2, C_3, \ldots, C_M]$ changes it to $[C_M, C_1, C_2, \ldots, C_{(M-1)}]$

Find whether you can transform the array B into the array A by applying the mentioned operation **exactly** once. If it is possible, find the **minimum** value of K for which it is possible.

EXPLANATION:

Z-function

Suppose we are given an array a of length n. The **Z-function** for this array is an array of length n where the i^{th} element is equal to the greatest number of elements starting from the position i that coincide with the first elements of a.

In other words, $z[i]$ is the length of the longest array that is, at the same time, a prefix of a and a prefix of the suffix of a starting at i.

If arrays A and B are same then minimum value of K is 0. From now on we assume that A and B are not same. Iterate on the array A from $i = 2\ to\ N$. If the minimum number of right-rotations, of any prefix of B, needed to make A and B same is $(i-1)$ then there must be a subarray of B which is same as the prefix of A of length $(i-1)$ with some restrictions on the starting point of such a subarray.
Let l and r be the left limit and right limit respectively for the starting point of such a subarray and let $SameFromEnd$ be the smallest index such that for all indices $i \geq SameFromEnd$, $A_i = B_i$ and len be the length of largest prefix of B which is same as the subarray of A of same length starting at index i.

Calculation of len for all indices of array A

The values of len for indices i are calculated in $O(N)$ time by applying Z-function on the array $(B + (-1) + A)$ where $'+'$ represents concatenation of the arrays.
Let zba be the array obtained after applying the Z-function on the array $(B + (-1) + A)$
then value of len for index i is zba_{n+1+i} .

Then l = $max(SameFromEnd - i + 1, 2)$, r = $len + 1$;

Left limit

The left limit is equal to $max(SameFromEnd - i + 1, 2)$ because if we select a subarray which starts with an index smaller than this limit, the prefix of length $(i-1)$ will be same but the suffix starting from index i of array A and B will not be the same as $A[SameFromEnd - 1]! = B[SameFromEnd - 1]$(from the definition of the variable $SameFromEnd$)

Right limit

The right limit is equal to $len + 1$ because if we select a subarray which starts with an index greater than this limit, the prefix of length $(i-1)$ will be same but the suffix starting from index i of array A and B will not be the

same as $A[i+len]! = B[len+1]$(from the definition of Z-function)

Now we just need to check effectively whether there is a subarray of B which is same as the prefix of A of length $(i-1)$. Let zab be the array obtained after applying the Z-function on the array $(A + -1 + B)$ where $'+'$ represents concatenation of the arrays. If the maximum value of the zab_i in the range $[n+1+l, n+1+r]$ is greater than or equal to $(i-1)$ then we have found a subarray and we can output $i-1$ as the minimum value of K.
For range query one can use any RMQ data-structure, modified to return maximum value in the range, like segment tree, sparse table etc.

For details of implementation please refer to the attached solutions.

TIME COMPLEXITY:

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

PROBLEM:

Initially, Chef is at coordinate 0 on X-axis. For each $i = 1, 2, \ldots, N$ in order, Chef does the following operation:

- If Chef is at a **non-negative** coordinate, he moves i steps backward (i.e, his position's coordinate decreases by i), otherwise he moves i steps forward (i.e, his position's coordinate increases by i).

You are given the integer N. Find the final position of Chef on the X-axis after N operations.

EXPLANATION:

Note the following:

1. Chef always starts at the origin.
2. i always begins at 1.
3. Chef's action depends on the previous state and i.

The above statements are enough to prove that chef's position is a function of i alone, i.e it can be written as position = $p(i)$.

$p(0) = 0$
$p(1) = -1$
$p(2) = 1$
$p(3) = -2$
$p(4) = 2$
$p(5) = -3$
$p(6) = 3$
and so on …

Note that:
At even $i's, p(i) = \frac{i}{2}$
At odd $i's, p(i) = \frac{-(i+1)}{2}$

Thus, Chef's position at $N = p(N)$.

$$p(N) = \begin{cases} \frac{N}{2} & if\ N\ is\ even \\ \frac{-(N+1)}{2} & if\ N\ is\ odd \end{cases}$$

TIME COMPLEXITY:

The above calculation can be done in constant time. Hence, the solution has a time complexity of $O(1)$.

PREREQUISITES:

Greedy

PROBLEM:

Given a body needs H units of continuous sleep. If you sleep $x(< H)$ units then next time, we need $2 * (H - x)$ units of continuous sleep.
Given a string of length L describing the day where $S_i = 0$ means that unit of time is free and $S_i = 1$ means that unit of time is occupied. Find if you would be able to achieve the sleep requirement.

EXPLANATION

As we know that if we sleep $x(< H)$ units then we need to sleep $2 * (H - x)$ units more next time.

So when we have some $x(< H)$ units of free time available then should us sleep or not?
We should sleep only if, it reduces our remaining sleep units that is we should sleep x units only if $2 * (H - x) < H$.
$=> 2 * H - 2 * x < H$
$=> 2 * H - H < 2 * x$
$=> H < 2 * x$
$=> x > H/2$

So iterate over the given string S, find $count$ of continuous $0s$, check if $count > H/2$, if yes then sleep for this time and update the H as following:
$H = 2 * (H - count)$

If at any point we have $H <= 0$ then we are able to achieve the requirement otherwise no.

TIME COMPLEXITY:

$O(L)$ per test case

PREREQUISITES:

Depth first search (dfs)

PROBLEM:

We are given a tree of N vertices and a string C of length N where each character can be R, G or B. We need to validate the input. The input is called valid if for all pairs of vertices (i,j) where $C_i = C_j = B$ we must have atleast one R vertex and atleast one G vertex in the shortest path between i and j.

EXPLANATION:

- Let us consider only the edges which are **not** of the type $R-G$ and construct a graph from it. We will end up with a forest of many connected components.
- Now we can observe clearly that if we consider one B vertex in connected component C_i and one one blue vertex from different connected component C_j, the path between them in the original tree will always have $R-G$ edge and hence those pairs are good. (After all, we ended up with connected components because of the removal of $R-G$ edges).
- Now what happens if we consider two B vertices in the same connected component ? I claim that if such a situation exists, then our answer is NO else our answer is YES. This is because if we consider the pair of B vertices with the shortest path in the component, it can be of the form $B-B$ or $B-R-R-R-\cdots-B$ or $B-G-G-G-\cdots-B$ which clearly has atleast one of the color R or G missing. This violates our condition and thus we shouldn't have more than one B vertex in a single connected component.
- The condition of checking number of B vertices in a connected component can be checked with a simple dfs traversal.

TIME COMPLEXITY:

$O(N)$ for each test case.

PROBLEM:

We are given four positive integers N, M, X, Y. We need to find the minimum cost to reach the point (N, M) from $(1, 1)$ if we can perform the following operations:

- Go down, up, left or right for cost X.
- Go diagonally down-left, down-right, up-right, up-left for cost Y.

QUICK EXPLANATION:

Basically we have three cases. Without loss of generality, assume $N \geq M$. The answer is the minimum of all of these cases:

- Reaching (N, M) by only using the first operation.
- Reaching the point (M, M) by only using the second operation and cover the remaining length by the first operation.
- Reaching the point (M, M) by only using the second operation. Now we can move consecutively in the following manner whenever possible i.e, move diagonally up-right and then down-right. Whenever this is not possible, use the first operation instead.

EXPLANATION:

Without loss of generality let us assume that $N \geq M$. (If it is not the case, we can swap N and M). This helps us simplify the problem without running into too many cases to handle. We need to go through these cases sequentially, if we are going through some case, then it means all the previous cases have failed to meet their conditions.

Case 1: $N = 1$ **or** $M = 1$

- Whenever this is the case, we can't move diagonally and hence can only use the first operation.
- Therefore, in this scenario the answer will be $(N + M - 2) \cdot X$.

Case 2: $Y < X$

- This condition gives us an intuition that we must use the second operation as frequently as possible.
- This condition means that, if we want to move from point (P, Q) to any of the points $(P + 2, Q), (P, Q + 2), (P - 2, Q), (P, Q - 2)$, instead of using the first operation two times, we can use the second operation two times.
- Based on this two ideas, one of the ways to construct an optimal path is to reach (N, M) in the following way: $(1, 1), (2, 2), \ldots (M, M), (M + 1, M - 1), (M + 2, M), (M + 3, M - 1), (M + 4, M) \ldots$
- But there is a catch here. We can end at two possible states if we follow the above procedure: at (N, M) or $(N, M - 1)$. This depends on the concept of **parity**. Initially we start at $(1, 1)$ where both x-coordinate and y-coordinate have the same parity. Whenever we use the second operation x-coordinate increments or decrements by 1 and the same thing happens for y-coordinate as well. Thus, if N and M have the same parity, we end up at (N, M). Else we end up at $(N, M - 1)$ and we need to apply one extra first operation to reach (N, M).
- Therefore the answer in this scenario will be $(M - 1 + (N - M)) \cdot Y = (N - 1) \cdot Y$ if the parities of N and M are same, or else the answer will be $(M - 1 + (N - M - 1)) \cdot Y + X = (N - 2) \cdot Y + X$.

Case 3: $Y < 2 \cdot X$

- In this case, moving one step diagonally to some location using the second operation has less cost than moving two steps using the first operation to that same location.
- Based on this idea, one of the ways to construct an optimal path is to reach (N, M) in the following way: $(1,1), (2,2), \ldots (M,M), (M+1,M), (M+2,M), (M+3,M), (M+4,M) \ldots$
- Therefore the answer in this scenario will be $(M-1) \cdot Y + (N-M) \cdot X$.

Case 4: $Y \geq 2 \cdot X$

- In this case, we need to use operations of the first type and reach (N, M) since any operation of the second type is costlier than the operation of the first type.
- Therefore the answer in this scenario will be $(N+M-2) \cdot X$.

TIME COMPLEXITY:

$O(1)$ for each testcase.

PREREQUISITES

Divide and Conquer, Interactive problems, Centroid of a tree.

PROBLEM

There is a tree consisting of N nodes. A certain node X is marked as special, but you don't know X - your task is to find it. To achieve this, you can ask queries to obtain information about X.

To be specific, you can ask queries in the form:

```
? Y
```

where $1 \leq Y \leq N$, and you will be provided with a random node on the path from node Y to node X, excluding Y. If $Y = X$ you will receive -1 instead.

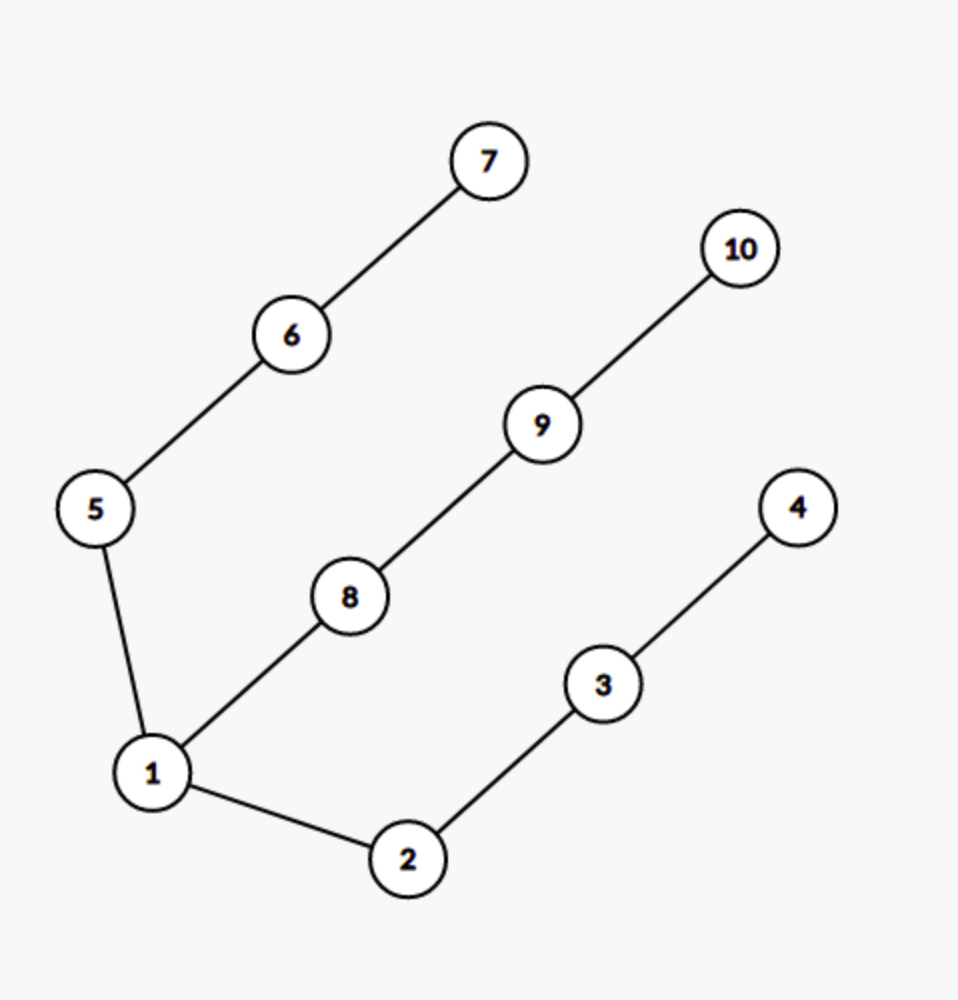
You can ask at most 12 queries. Find the special node X.

QUICK EXPLANATION

- At each step, query for the centroid of the tree. This way, the response to the query shall always be a subtree half the size of the original tree.
- When you query a node q and receive a response v, you only need to consider the subtree which contains the response node v.
- To identify which subtree node v belongs to, you can run BFS or DFS. We can ignore the rest of the tree.

EXPLANATION

We'll consider the following tree throughout the editorial



Let's assume we query at node 1.

- If the hidden node is 1, the case is solved.
- If the hidden node is among $[2, 3, 4]$, we'd get a node from $[2, 3, 4]$ in response.
- If the hidden node is among $[5, 6, 7]$, we'd get a node from $[5, 6, 7]$ in response.
- If the hidden node is among $[8, 9, 10]$, we'd get a node from $[8, 9, 10]$ in response.

Hence, based on the response, we are able to reduce the possible candidates for X from 10 to 3.

Let's assume we query at node 8 instead of 1.

- If the hidden node is 8, the case is solved.
- If the hidden node is among $[9, 10]$, we'd get a node from $[9, 10]$ in response.
- If the hidden node is among $[1, 2, 3, 4, 5, 6, 7]$, we'd get a node from $[1, 2, 3, 4, 5, 6, 7]$ response.

In this case, assuming the worst, we are able to reduce the possible number of candidates from 10 to 7 (happens when the response is 1 for querying node 8).

So, it is better to query node 1 here as compared to 8.

Observation

We want to query at a node such that the size of the largest subtree of its child, is minimized. For node 1, it had 3 children of size 3 each, while node 8 had two children, one of size 2 and one of subtree size 7.

Claim: It is always possible to reduce the number of candidates by at least half in each query.

Anyone, who has used **centroid decomposition** even once would immediately know that centroid of a tree is a node such that no subtree of this node has a size greater than the size of the original tree.

Hence, for each query, we shall query the centroid of the remaining tree, find the subtree to which the response node belongs, and discard the rest of the tree.

For example, if when queried node 1, if we receive node 7 in response, we only need to keep tree consisting of nodes $[5, 6, 7]$.

Why doesn't this exceed 12 queries?

At each query, the number of candidates reduces by at least half. At the start, there are N candidates. At the end, in order to solve the problem, there must be only one candidate left.

Hence, the number of queries must be the smallest integer x such that $\lfloor \frac{N}{2^x} \rfloor \leq 1$ which implies $N \leq 2^x \implies x \geq log_2(N)$.

For $N \leq 1000$, this translates to roughly 10 or 11 queries. depending upon implementation.

TIME COMPLEXITY

The time complexity is $O(N * log(N))$ or $O(N^2)$ depending upon implementation

PREREQUISITES:

Binary Lifting

PROBLEM:

Consider the following string of infinite length: `abc...xyzabc...xyz...`

Let string B be the prefix of this string of length M. For e.g. if M=6, B= `abcdef`.

Vedansh calls a string X of length K special if it satisfies the following conditions:

- B is a substring of X. Formally, for some $1 \leq L \leq R \leq K, X_{L...R} = B$

Vedansh has string A of length N (consisting of lowercase Latin letters only) and he wants to convert it into a special string. To do so he can perform the following operation:

- Pick an i such that $1 \leq i \leq |A|$ and delete A_i. The remaining parts of A are concatenated.

Since Vedansh is lazy, he wants to do this in minimum number of operations or determine if it can not be converted to a special string. Help Vedansh in doing so.

QUICK EXPLANATION:

What if we fix the position of the first character of B in X?

We will pick the remaining characters greedily by choosing the next character at its next occurrence. And thereby, the subsequence corresponding to B gets fixed. We can just delete the unwanted characters between the first and last character of B.

How to check optimally for all starting positions?

If we keep picking the next occurrence in X of the next character of B, we'll have to make O(n) computations each time. To do this optimally we'll have to store some information prior to this step.
We can store the position of later characters in multiple jumps.

Which technique and data structure to use to perform these operations optimally?

The binary lifting technique and the corresponding dp values provide this functionality to check positions of characters in jumps of 1,2,4 and so on. We can use this to find the position of character after M-1 positions and then pick the result that uses the minimum number of delete operations.

EXPLANATION:

The string B will appear in string X as a substring if there is a subsequence B in the string X. So corresponding to a particular subsequence B, we will have the delete the unwanted characters in between the first and last character to obtain B as a substring. We have to pick such a subsequence in which the difference between the position of the first and last character is minimum, thereby leading to the minimum deletions from X.

If we fix the position of the first character of B in X(in this case 'a'), the next character should be chosen as near as possible to minimize the deleted characters. If we proceed in the above manner the whole subsequence gets fixed and hence the position of characters of B.
So we want to fix the first character at every position possible and check the difference b/w the first and last character of the resultant subsequence and pick the one with the least difference, hence the minimum deletions. If we carry out these operations individually and directly we would require O(n) time at each step giving time limit errors.

To carry out these operations optimally we have to make some calculations and store some info prior to this step. In the naive approach, we calculate the position of the next element every time and repeat this step. By invoking our knowledge of binary lifting, we try to calculate the position of the next to next element, the 4th next element, and so on. This can be easily calculated using the dynamic programming technique.

After storing these values we can calculate the (m-1)th next element from each starting point using the binary lifting technique and pick the subsequence that requires the minimum deletions.

TIME COMPLEXITY:

O(nlogn)

PREREQUISITES:

Observation, Prime Factorisation, Smallest Prime Factor

PROBLEM:

Let's define a function $F(X)$ as follows:

$$F(X) = \frac{X}{Y}$$

where Y is the largest perfect square that divides X.

You are given an array A consisting of N integers. A pair of integer (i, j) is called *Good* if $1 \leq i < j \leq N$ and $F(A_i * A_j) > 1$. Find the number of *Good* pairs.

EXPLANATION:

The first and basic observation is that $F(X)$ is always greater than 1 if X is not a perfect square. When X is a perfect square then Y is equal to X and hence $F(X)$ becomes equal to 1. In all other cases, Y will be always less than X, and hence the value of $F(X)$ will be greater than 1.

So ultimately our goal is reduced to count such pairs (i,j) where $A_i * A_j$ is not a perfect square. All such pairs will have $F(A_i * A_j)$ greater than 1.

Let's try to go opposite and try to find all such pairs (i,j) where $A_i * A_j$ is a perfect square. After getting these pairs it is easy to find out those pairs where $A_i * A_j$ is not a perfect square.

Now let's see how we will be able to find all such pairs. Suppose we have two numbers A_i and A_j whose prime factorization looks like:

$$A_i = a^x * b^y$$

$$A_j = c^i * d^j$$

Now let's look at the conditions when $A_i * A_j$ is a perfect square:

- If i, j, x and y are even, then the product is always a perfect square.
- If some of the powers in A_i are odd then the same base should be present in A_j with power as odd. More formally if x is odd then either $c = a$ or $d = a$ with i and j as odd.

Hence we can divide our groups on the basis of those prime numbers that divide the integer odd number of times. For example, let us see the sample:

$A = [2, 3, 12]$

Now:

- 2 is divisible by 2 odd number of times
- 3 is divisible by 3 odd number of times
- 12 is divisible by 3 odd number of times

Then there will be two groups:

- **Group 1**: $[2]$
- **Group 2**: $[3, 12]$

Now we can say that all the elements in the group will form pairs that have $A_i * A_j$ as a perfect square. Hence we can easily count the total number of groups that have $A_i * A_j$ as a perfect square

Since the value of N and T are large we can precompute for every integer I to which group it belongs, using the technique of Smallest Prime Factor.

Once we got the count of groups we can then simply found the total pairs where $F(A_i * A_j) > 1$.

TIME COMPLEXITY:

$O(N * log(N))$ per test case

PREREQUISITES:

SOS Dynamic Programming

PROBLEM:

Chef has an array A of N integers. He asks you Q queries. For each query, you are given the following:

- An integer X $(0 \leq X < 2^{20})$.
- An array B of size M such that $1 \leq B_j \leq N$.

You need to erase **minimum** number of elements from A such that the bitwise-OR of the remaining elements is **at most** X and no element having index B_j is erased.

For each query, find the **maximum** number of remaining elements such that the bitwise OR of the remaining elements is **at most** X and no element with index B_j is erased. If it is impossible to get an OR of at most X, print -1.

Note: The bitwise OR of an empty array is considered 0.

QUICK EXPLANATION:

- Let us calculate $dp[mask]$ which denotes the number of elements A_i such that $A_i \& mask = A_i$ using SOS Dynamic Programming
- Subtask-1: We can iterate over all *mask* such that $0 \leq mask \leq X$, and take the maximum over all $dp[mask]$.
- Subtask-2: We can iterate from the most significant bit towards lowest significant bit and try calculating answer at each step.

EXPLANATION:

Subtask-1

To make explanation more lucid, let us first define subset of X as following: An element Y is called a subset of X if the set-bits in the binary representation of Y is a subset of set-bits in the binary representation of X. So, If $X = 1010_2$, then $Y = 1010_2, 1000_2, 0010_2, 0000_2$ represents subset of X.

Let us first try to answer a simpler version of Subtask-1. Suppose we are given an integer X, and we are asked to find out the maximum number of elements whose bitwise-OR is a subset of X.
So, if A_i is one of the elements which forms the answer, then we can say that $A_i \& X = A_i$. In other words, we want to find out the the number of elements A_i such that $A_i \& X = A_i$. Using SOS Dynamic Programming we can calculate the answer for each possible X and store it in $dp[X]$.

Now, once we have this dp calculated, we can try solving our original Subtask-1. We are given X, and we want to find out the maximum number of elements such that their bitwise-OR is at most X. Note that all the subsets of X are less than or equal to X, hence if we take the max over all $dp[i]$ such that $0 \leq i \leq X$, we will get our required answer!

Subtask-2
Now we are also given some elements which cannot be removed from the array. Let us denote their bitwise-OR by Y. If $Y > X$, then we can say that the answer is -1, otherwise we can start by taking our current answer as $dp[Y]$.

Now let's start with the most significant bit, and iterate towards the lower significant bit. At each point, we will consider the cases if this bit is set in the final OR or not, and will update our answer accordingly. Let the current

bit be the i^{th} bit. We have the following two cases:

The i-th bit is set in X

Let val denotes the current value such that the final OR is a subset of val and | denotes the bitwise-OR.

If $val|2^i > X$, then this means that we cannot set this bit in our final OR, as it will make it greater than X.

Otherwise, we have the freedom to either set this bit in our final answer, or let it remain unset.
If we don't want to set this bit, then we can set any of the bits at positions 0 to $i-1$ (as the resultant OR will always be less than X), and therefore we can update our answer using the dp that we calculated ($max(ans, dp[val'])$, where val' has all the bits set at positions 0 to $i-1$, unset at position i, and same as val for positions greater than i)
If we want to set this bit, we can update our $val \rightarrow val|2^i$, and update our answer as $max(ans, dp[val])$

The i-th bit is not set in X

In this case, this bit cannot be set in the final OR, as that will make the final OR greater than X. Hence, we ignore this bit, and continue

TIME COMPLEXITY:

To calculate the dp, we will require $O(N \cdot 2^N)$ time, where $N = 20$ in our problem.
For each query, we will require $O(N + M)$ time.

PREREQUISITES:

Z Function, Prefix Sums, RMQ

PROBLEM:

Given three strings S_1, S_2 and X, calculate the number of non-empty substrings of X that can be expressed as $P + Q$, where P and Q are prefixes of S_1 and S_2 respectively.

EXPLANATION:

Call a non-empty substring of X *special* if it can be expressed as the concatenation of some prefix of S_1 and S_2. Then, we want to find the number of special substrings of X.

Claim: If $X[L..R]$ is special, then $X[L..R']$ is also special, for all $L \leq R' \leq R$.

Proof

Let $X[L..R]$ be expressed as $P + Q$, where P and Q are prefixes of S_1 and S_2 respectively.

To get $X[L..R']$, we can keep erasing the last element of Q (erase from P if Q becomes empty) till $P + Q$ equals $X[L..R']$. All the while, P and Q still remain prefixes of S_1 and S_2, implying that $X[L..R']$ is also special.

Thus, proved.

Thus, if for each valid L, we can find the largest R_L such that $X[L..R_L]$ is special, the answer to the problem is equivalent to

$$\sum(R_L - L + 1)$$

***Note:** P and Q in the rest of the editorial exclusively represents some prefix substring of S_1 and S_2, respectively.*

Consider some *special* substring $X[L..R] = P + Q$. If we want to maximize R for a fixed L, the combined length of P and Q should be maximised.
Then, for a fixed L, we can try each possible value of P and, greedily maximising the corresponding length of Q in each case, calculate the maximum possible combined length of P and Q.

Example

Let $S_1 = \text{aab}, S_2 = \text{bbc}, X = \text{aabbc}$.
For fixed $L = 1$, the possible P and **longest possible** Q are:

- $P =$ “aab", $Q =$ “b"
- $P =$ “aa", $Q =$ “bbc"
- $P =$ “a", $Q =$ “"
- $P =$ “", $Q =$ “"

The maximum possible combined length is then 5 (case 2 above).

A thing to observe here is that - if P' is the longest possible P, then each valid P is a prefix of P'. This trivial observation motivates the solution.

Define $Z_1[i]$ as the length of the longest prefix of S_1 that is a prefix of the suffix of X ending at index i. Define $Z_2[i]$ for S_2 similarly.

How do I calculate this?

Both arrays can be calculated in linear time using the **Z function**.

If the Z-array of string $S_1 + \text{“\#"} + X$ is Z, then it is clear that array Z_1 is equal to the length $|X|$ suffix array of Z. Similarly for Z_2 too.

Then, for a fixed L, the maximum possible length of $P + Q$ is equal to

$$\begin{gathered}\max(Z_1[L] + Z_2[L + Z_1[L]],\\ (Z_1[L] - 1) + Z_2[L + Z_1[L] - 1],\\ (Z_1[L] - 2) + Z_2[L + Z_1[L] - 2],\\ \vdots\\ (0) + Z_2[L])\end{gathered}$$

which can be written compactly as

$$\begin{aligned}&\max_{0 \le K \le Z_1[L]} \{(Z_1[L] - K) + Z_2[L + Z_1[L] - K]\}\\ &= Z_1[L] + \max_{0 \le K \le Z_1[L]} \{Z_2[L + Z_1[L] - K] - K\}\end{aligned}$$

This value can be computed quickly using RMQ with suitable preprocessing.

How?

Define array B such that $B[i] = Z_2[i] + i$. Then the above equation can be rewritten as:

$$\begin{aligned}&Z_1[L] + \max_{0 \le K \le Z_1[L]} \{B[L + Z_1[L] - K] - (L + Z_1[L])\}\\ &= Z_1[L] + \max_{0 \le K \le Z_1[L]} \{B[L + Z_1[L] - K]\} - (L + Z_1[L])\end{aligned}$$

which is further simplified to get

$$\max_{0 \le K \le Z_1[L]} \{B[L + Z_1[L] - K]\} - L$$

The problem is now reduced to finding the Range Maximum on static array B, which can be **done efficiently using a Sparse Table**.

Using this, we can determine R_L for each L and compute the answer of the problem (using the formula at the end of the first section of the editorial).

TIME COMPLEXITY:

Computing Z_1 and Z_2 takes $O(|S_1| + |X|)$ and $O(|S_2| + |X|)$ respectively. Precomputing the sparse table for RMQ takes $O(|X| \log |X|)$. With constant time RMQ, computing R_L for each L takes $O(1)$.
The total complexity is therefore

$$\begin{gathered}O(|S_1| + |X| + |S_2| + |X| + |X| \log |X| + |X|) \approx\\ O(|S_1| + |S_2| + |X| \log |X|)\end{gathered}$$

per test case.

PROBLEM:

An array A of size N is called stable if all the elements in it are equal. We need to find the minimum number of operations to convert an array into a stable array if an operation can be any one of the following:

- Select prefix $A[1, 2 \ldots, k]$ where $A_k >= A_i$ for all $1 \leq i \leq k$ and perform $A_i = A_k$ for all $1 \leq i \leq k$.
- Select any segment $A[L, \ldots, R]$ and cyclically shift the elements in this segment to the left keeping other elements in the array unchanged.

EXPLANATION:

- If initially all the elements in the array are equal, obviously the answer is 0.
- Let max be the maximum element in the array and let it be at index ind. (If multiple indices are possible, take ind as the maximum among them).
- If $ind = N$, we can just perform the first operation taking $k = N$ and convert all the elements in the array equal to max.
- If $ind < N$, we need to perform a total of 2 operations. First, we need to apply operation 2 on the segment $A[ind, \ldots, N]$. This sends the value at index ind (i.e, max) to index N. Now we can perform operation 1 taking $k = N$, and convert all the elements in the array equal to max.

TIME COMPLEXITY:

$O(N)$ for each testcase.

PREREQUISITES:

Greedy Algorithm

PROBLEM:

Initially, Alice and Bob both have a bag of N stones each with every stone having a **lowercase alphabet** written on it. Both players know the stones of the other player.

Alice and Bob want to create a string S of length $2 \cdot N$ using the stones they have. They play a game to create the string. The rules of the game are as follows:

- Initially, all the indices of S are empty.
- Alice starts the game. The players take alternating turns. In each turn, the player can choose one of the stones from their bag and place it at any empty index i $(1 \leq i \leq 2 \cdot N)$ of S.
- The game ends when both the players have used all their stones.
- Alice wants S to be as lexicographically small as possible while Bob wants S to be as lexicographically large as possible.

Find the final string S formed by characters written on stones if both Alice and Bob play optimally!

EXPLANATION:

Observation 1

Alice wants the string to be as lexicographically small as possible.

For each turn, Alice would want the smallest empty index to be filled with the lexicographically **smallest** letter available. Similarly, she would want the largest empty index to be filled with the lexicographically **largest** letter available.

On the other hand, Bob would want the smallest empty index to be filled with the lexicographically **largest** letter available and the largest empty index to be filled with the lexicographically **smallest** letter available.

Observation 2

Consider the case when all the letters Alice has, are lexicographically larger than the letters of Bob.

The smallest letter Alice chooses would be larger than any letter chosen by Bob.

Thus, it is not optimal for Alice to fill the smallest empty index with the smallest letter available to her. However, since all her letters are larger than Bob's, she can fill the largest empty index with the largest letter available to her.

Similarly, if all the letters of Bob are lexicographically smaller than the letters of Alice, it is not optimal for Bob to fill the smallest empty index with the largest letter available to him. Instead, he can fill the largest empty index with the smallest letter available to him.

Solution
Optimal moves for Alice

- Place lexicographically **smaller** letters at smaller empty indices.
- Place lexicographically **larger** letters at larger empty indices.

Optimal moves for Bob

- Place lexicographically **larger** letters at smaller empty indices.
- Place lexicographically **smaller** letters at larger empty indices.

Suppose it is Alice's turn to place a stone.

From all the stones available to her, she chooses the stone with lexicographically **smallest** letter.

Suppose this letter is lexicographically greater than the largest letter available to Bob. This means that all letters of Alice are larger than Bob's . Thus, Alice has the overall lexicographically **largest** character available amongst **both** of them. She can place the lexicographically **largest** character at the largest empty index.

On the other hand, if this not the case, she can place the lexicographically **smallest** character **available to her** at the smallest empty index.

Similarly, if it is Bob's turn:

From all the stones available to him, he chooses the stone with lexicographically **largest** letter.

Suppose this letter is lexicographically smaller than the smallest letter available to Alice. This means that all of Bob's letters are smaller than Alice's . Thus, Bob has the overall lexicographically **smallest** character available amongst **both** of them. He can place the lexicographically **smallest** character at the largest empty index.

On the other hand, if this not the case, he can place the lexicographically **largest** character **available to him** at the smallest empty index.

Note that when the largest letter of Bob is **equal** to the smallest letter of Alice, the optimal move for any player would be to place the letter at the **largest** empty index.

TIME COMPLEXITY:

Based on the implementation, the time complexity can be $O(Nlog(N))$ or $O(N)$ per test case. Refer Editorialist's Solution for $O(N)$ implementation.

PROBLEM:

Chef has a string S consisting only of English lowercase letters (a - z). However, Hitesh does not like it and wants it to be reversed.

Hitesh wonders what is the minimum number of operations required to reverse the string S using the following operation:

- Select any i such that $1 \leq i \leq |S|$ and remove S_i from its original position and append it to the end of the string (i.e. shift any character to the end of the string).

For e.g. if $S = \texttt{abcde}$ and we apply operation on $i = 2$, then S becomes $\texttt{acdeb}$.

Help Hitesh find the minimum number of operations required to reverse S.

It is guaranteed that it is possible to reverse the string in a finite number of operations.

QUICK EXPLANATION:

- In the optimal scenario, we will never remove the same character twice. In this sentence, we are treating each index element as a distinct character. So, in aba, the first and the third a are distinct for this sentence.
- Let S' denote the set of indices (in the original string) that we have removed and appended at the end, and let S denote the set of indices that we have not operated on. How will the string look after all the operations.
- Make the cardinality of S as large as possible.

EXPLANATION:

The first observation is in the optimal scenario, we should never remove the same character twice. To see this, consider a sequence of operations O_1 in which the same character c is removed twice. Consider another sequence of operations O_2, where the first operation in which the character c was removes, is removed from O_1 , and rest everything remains same. We can see that both O_1 and O_2 will lead to same string, with $|O_2| < |O_1|$.

Let $S = \{i_1, i_2, ..., i_k\}$ be the set of indices which have not been operated on. Let us call the new string as new_str. We have:

$str = s_1 s_2 ... s_n$
$rev\_str = s_n s_{n-1} ... s_2 s_1$
$new\_str = s_{i_1} s_{i_2} ... s_{i_k}\ s_{i'_1} ...,$

Because $rev\_str = new\_str$, we have $s_n = s_{i_1}, s_{n-1} = s_{i_2}$, and so on. So to minimize the size of set S', we should maximize the size of set S, which is equal to k. In other words, we need to find out the longest prefix of rev_str that appears as a subsequence in str.

We can find this prefix greedily, using the algorithm described **here** in $O(N)$ time.

TIME COMPLEXITY:

$O(N)$ per test case.

PREREQUISITES

Combinatorial game theory, winning and losing states in game theory

PROBLEM

Given a binary string S of length N, Alice and Bob play on this string, taking turns to apply the operation. The player unable to apply operation loses.

In an operation, the player chooses an index $1 \leq i < |S|$ such that $S_i \neq S_{i+1}$ and erase either S_i or S_{i+1}. The remaining parts after erasing character are joined and become adjacent.

QUICK EXPLANATION

- If the string contains only zeros or only ones, Alice cannot make a move at all, so Bob wins
- If the string contains exactly one occurrence of zero (or exactly one occurrence of one), then Alice can delete that one character, putting Bob in losing position.
- Otherwise, we can prove that the answer depends on the parity of N. Alice wins only if N is odd.

EXPLANATION

Since this is a game theory problem, understanding of winning and losing states is crucial here.
Refer this before reading further.

Let us tackle the easier cases. If the string consists of all zeros or all ones, no operation is possible, so it is losing position.

If the string contains exactly one occurrence of one of the characters, then Alice can delete that unique character, putting Bob in a losing position. Thus, if either 0 or 1 appears exactly once, Alice wins.

General case

Let's denote c_0 as the number of 0 in S, and c_1 as the number of 1 in S.

Now we can assume that $c_0, c_1 \geq 2$. In one operation, a player can delete either one occurrence of 0 or one occurrence of 1.

Let's denote (c_0, c_1) as the state of string S, having c_0 zeros and c_1 ones. Then from state (c_0, c_1), we can only go to state $(c_0 - 1, c_1)$ or $(c_0, c_1 - 1)$. We can visualize it as a DAG of states.

For each state, we aim to determine whether it is a winning state or a losing state.

From base cases, we know that states $(x, 0)$ and $(0, x)$ are losing state for any x and states $(1, x)$ and $(x, 1)$ are winning states for any x.

What happens for state $(2, 2)$, we can go to state $(1, 2)$ or $(2, 1)$, both of which are winning states. Hence, we cannot put our opponent in losing state. Hence, $(2, 2)$ is a losing state.

Similarly, if we are on state $(2, 3)$, we can put our opponent to $(1, 3)$ or $(2, 2)$. We know that $(1, 3)$ is winning, but $(2, 2)$ is losing. Hence, we can put our opponent to losing state by moving to $(2, 2)$. Hence, $(2, 3)$ is the winning state.

Proceeding like this, we get the following matrix where cell (i, j) denote whether state (i, j) is winning or losing.

```
00000000
01111111
01010101
```

```
01101010
01010101
01101010
01010101
```

It is easy to prove that except for base cases, the answer depends only on the parity of N. The state (x, y) is the winning state if and only if $x + y = N$ is odd.

TIME COMPLEXITY

The time complexity is $O(|S|)$ per test case.

PROBLEM:

*This problem is similar to the problem "STRFLIP2". The only difference between them is the number of moves allowed — in this problem, up to N moves can be made.

You are given a binary string A of length N, which we treat as being 1-indexed.

In one move, you are allowed to

- Pick a **contiguous** substring (i.e. $A[L:R]$ for some $1 \leq L \leq R \leq N$) of A that contains **at least one** '0' and **at least one** '1'
- Flip all the characters of the chosen substring (i.e. change every 0 to 1 and every 1 to 0)

You are also given a target string B of the same length as A.

Determine if it is possible to convert A to B by applying a sequence of moves as described above.

If it is possible to convert A to B, then find a sequence of no more than N moves that accomplishes this.
It can be shown that whenever it is possible to convert A to B, then it is possible to do so using no more than N moves.

If there are multiple answers, you can print any of them.

QUICK EXPLANATION:

First Observation

Let's assume A is not equal to B.
We cannot apply any operation on the string A if it contains all $0's$ or all $1's$. Similarly, if B contains all $0's$ or all $1's$, we can never convert A into B.

When is it impossible to convert A to B?

Extending our observation 1 we can definitely say that if either A or B contains all $0's$ or all $1's$ and they are not equal then we cannot make them equal. Suppose both A and B contains at least one 0 and at least one 1. Can we always convert A to B? The answer is Yes!
Let's look at the following approach.

Approach for N operations

Since we have N operations it looks like we can fix one index in each operation. Lets say we have some index x and $x+1$ such that $A[x] \neq A[x+1]$.
Now using these two indices, we can first fix the indices $[1...(x-1)]$, and $[(x+2)...N]$ (by always including indices x and $x+1$ in the chosen substring).
Finally, we will try to fix $A[x]$ and $A[x+1]$.

What should be the value of x

Note that at the end, we want to fix $A[x]$ and $A[x+1]$. As we know, $A[x] \neq A[x+1]$, it would be good if $B[x] \neq B[x+1]$, so that we can easily fix both x and $x+1$ in at max one single operation.

So, we want to find such $x : B[x] \neq B[x+1]$. If $A[x] = A[x+1]$ for this x, then we can choose one of $A[1...x]$, $A[(x+1)...N]$, and flip this substring to make $A[x] \neq A[x+1]$

Approach for N/2+10 operations

Term $N/2$ suggests that we should fix 2 indices in each operation. Lets try to modify the above approach.

Instead of using x and $x+1$ indices, let's try to use $N/2$ and $N/2+1$ to make the whole string equal.

Now instead of fixing one index we will try to fix one index in left half (i.e. $[1...N/2-1]$) and one index in right half (i.e. $[N/2+2...N]$.

Lets say L is the leftmost index such that $A[L] \neq B[L]$ (Let $L = N/2$ if there is no mismatching index in left half).

Similarly R is the rightmost index such that $A[R] \neq B[R]$ (Let $R = N/2+1$ if there is no mismatching index in right half).

Now flipping $[L, R]$ will fix two indexes (L and R). So after approximately $N/2$ operations we would have fixed every index apart from $N/2$ and $N/2+1$.

Fixing N/2 and N/2+1

Let x be any index such that $B[x] \neq B[x+1]$.

If $x = N/2$ then flipping $\{N/2, N/2+1\}$ will do our job.

Now let $x < N/2$, then flipping $[x...(N/2+1)]$ and $[x...N/2]$ would make values of $A[N/2] = A[N/2+1]$. Now we can fix x and $x+1$ which could be easily fixed.

EXPLANATION:

TIME COMPLEXITY:

SOLUTION:

Tester's Solution - STRFLIP

Tester's Solution - STRFLIP2

Setter's Solution - STRFLIP

Quote

---

suzan1234 **Jan '22**

It would have been much better if the code provided was free from macros

---

codersachit **Jan '22**

nice “constructive” round

---

gumnami **Jan '22**

For the input:
1
4
0111
1001
The editorial codes give the output:
1
1 3

How is this valid?
Shouldn’t the answer be “-1”?

---

coderalways1 **Jan '22**

Power of the editorialist maybe?!

---

ghostofuchiha **Jan '22**

In the given constraints the minimum length of given strings are 20 . So its not a valid input but still the answer is correct try to do the operation on paper!

PREREQUISITES:

Bitwise Xor

PROBLEM:

JJ is back with another challenge. He challenges you to construct an array A containing N integers such that the following conditions hold:

- For all $1 \leq i \leq N$, $1 \leq A_i \leq 10^6$
- Every subarray has non-zero XOR. That is, for every $1 \leq L \leq R \leq N$, $A_L \oplus A_{L+1} \oplus \ldots \oplus A_R \neq 0$. Here, $\oplus$ denotes the bitwise XOR operation.

Can you complete JJ's challenge?

Under the given constraints, it can be proved that there always exists at least one array satisfying these conditions. If there are multiple possible arrays, **print any of them**.

QUICK EXPLANATION:

What if there is a subarray having XOR = 0?

If there is a subarray having XOR $= 0$, there will exist a pair of distinct values i, j such that the XOR of prefixes of length i and j will be equal.
Hence, if XOR of all prefixes are distinct, then there will be no subarray having XOR $= 0$.

Assigning XORs

So let us assign distinct positive XORs to each prefix length. One way can be to assign $XOR_{[1,i]} = i$.
This means that $(i-1) \oplus A_i = i$

EXPLANATION:

$\oplus_{[L,R]}$ denotes the XOR of the subarray starting at index L and ending at index R (inclusive).

Let's say there exists a subarray $[L, R]$ such that $A_L \oplus A_{L+1} \oplus \ldots \oplus A_R = 0$
Consider the XOR of the prefix $[1, L-1]$ and $[1, R]$.

$$\oplus_{[1,R]} = (\oplus_{[1,L-1]}) \oplus (\oplus_{[L,R]}) \implies \oplus_{[1,R]} = \oplus_{[1,L-1]}$$

The above equation suggests that if there is a subarray having XOR $= 0$, there will exist a pair of distinct values i, j such that the XOR of prefixes of length i and j will be equal.

Hence, if XOR of all prefixes are distinct, then there will be no subarray having XOR $= 0$.

Now, there can be several ways in which we can assign the XOR values to each prefix length. One of the simplest way is to assign $\oplus_{[1,i]} = i$, i.e. for prefix of length i, let us assign it's XOR value to be i.

Getting the array

So we have $\oplus_{[1,i]} = i, \forall i : 1 \leq i \leq N$
XOR of prefix of length one $= 1 = A_1$

Now, for a general index $i > 1$, $\oplus_{[1,i]} = \oplus_{[1,i-1]} \oplus A_i$
Substituting values, $i = (i-1) \oplus A_i \implies A_i = (i-1) \oplus i$

It can be proved that the above value of A_i ensures that $A_i < 2 \cdot i$. Therefore, $\forall i : 1 \leq i \leq N, A_i \leq 10^6$

Bonus Problem

Suppose we modify the constraint of A_i, such that $1 \leq A_i \leq N$. Can you construct an array now?

TIME COMPLEXITY:

Assuming that taking XOR of two numbers is an $O(1)$ operation, the above approach will take $O(N)$ time for each testcase.

PREREQUISITES:

Tree, Graph Theory, LCA

PROBLEM:

You are given a tree with N nodes (numbered 1 through N) and Q queries (numbered 1 through Q). For each valid i, in the i-th query, you are given K_i nodes $x_1, x_2, \ldots, x_{K_i}$. Consider the smallest subtree which contains all of these nodes; you should find all centers of this subtree.

A node is called a *center* of a tree if it lies in the middle of at least one longest path in that tree. Note that there may be multiple longest paths (paths with the same maximum length) and for a longest path which contains an even number of nodes, there are two nodes lying in the middle of this path.

Note: A subtree here refers to a connected subgraph of the tree. Selecting a node does not mean all its descendants have to be selected.

EXPLANATION:

As we are concerned about the minimal subtree, it means that all the leaves of this subtree are the nodes that are in the given query. Since selecting descendants of those query nodes is not optimal as it increases the size of the subtree.

The diameter of any tree is the longest path of the tree. Since diameter always exists between two leaves in the tree and all the leaves of our minimal subtree are query nodes. Hence we just need to find the pair of nodes from the query nodes that has the maximum distance between among all such pairs.

What happens when there is more than one such pair?

There can be multiple pairs of nodes whose distance is equal to the longest path of the minimal subtree. However, the center of all such pairs remains the same.

- Let's say there are two pairs (a, b) and (c, d), such that both pairs have a distance equal to the longest path of the tree.
  - It is quite clear that the $lca(a, b) = lca(c, d)$. If it is not so then both of these pairs cannot be equal to the longest path of the tree. As we can take one node from pair (a, b), say b $(depth[a] < depth[b])$ and similarly one node from pair (c, d), say d $(depth[c] < depth[d])$. Then the pair (b, d) will have the distance which is longer than both of the pairs.
  - If both pairs of a node have the same lca, then the center of all such pairs which has the longest path will be the same. To find the longest path we select two such leaves which are farthest from the lca, say x and y. If $x = y$, then the center will be the lca, else the center lies on the path from lca to x.

Now, we can easily found the end nodes of the diameter, and finally, we can find the center of the longest path.

One of the end nodes is the query node which is at the maximum depth of the tree. And to find the other end node of the diameter we can find the distance of the first node with every other node and the node which has maximum distance is our another end node. The distance between two nodes can be found by using the lca technique. To find lca we can use the binary lifting technique.

TIME COMPLEXITY:

$O(N * log(N) + sumK * log(N) + Q * log(N))$

PREREQUISITES:

MEX of a set
The MEX of an array is the minimum non-negative integer that is not present in it.

PROBLEM:

Chef was on vacation when a thought struck him. Given three integers N, K, and X, he would like to make an array A of length N such that it satisfies the following conditions:

- $0 \leq A_i \leq N$ for each $1 \leq i \leq N$
- The MEX of every contiguous **subarray** of length K in A is X.

Please help Chef in finding such an array. If there are multiple answers, you can output **any** configuration; and if there is no possible answer, output -1.

Note: The MEX of an array is the minimum non-negative integer that is not present in it. For example,

- The MEX of array $[0, 2, 5, 1]$ is 3, because it contains $0, 1$ and 2 but not 3.
- The MEX of array $[1, 2, 5]$ is 0, because 0 is the smallest non-negative integer not present in the array.
- The MEX of array $[0, 1, 2, 3]$ is 4.

QUICK EXPLANATION:

When will the answer be -1?

If $X > K$, the answer will be -1. A valid array will always exist if $X \leq K$.

The first K elements

If $X \leq K$, we want first K elements to have all the values from 0 to $X - 1$ (inclusive). There can be several ways to do this. One of the ways is to choose all the values from 0 to K (inclusive) except X.

Next elements

Start with analyzing the value of A_{K+1}. You will soon realize that one of the ways is to have $A_i = A_{i-K}$.

EXPLANATION:

Let us first consider the case when the answer will be -1. For a given subarray of length K, what can be the maximum value of MEX? The answer is K, when all the elements from 0 to $K - 1$ appears in the subarray. This means that the answer will be -1 if $X > K$.

We will now show that a valid array will always exist if $X \leq K$ through construction.

Consider the first subarray of length K, i.e. the first K elements. We want the first K elements to have all the values from 0 to $X - 1$ (inclusive). There can be several ways to do this. One of the ways is to choose all the values from 0 to K (inclusive) except X.

So now, we have fixed our $\{A_1, A_2, ..., A_K\}$. Consider the next K elements, i.e. $\{A_2, A_3, ..., A_{K+1}\}$. We want the MEX of these elements to be X. Let us restate our current problem:
We know that MEX of $\{A_1, A_2 \cdots A_K\}$ is X. We want to add one element to the set $\{A_2, A_3, ..., A_K\}$ such that it's MEX will become X. What should be that element? We can see that adding A_1 to this set will obviously make the MEX of the set to be X. Hence, substituting $A_{K+1} = A_1$ solves our current problem!

Continuing in the similar way, we can see that substituting $A_i = A_{i-K}$ will solve our problem!

TIME COMPLEXITY:

The time complexity of the above approach will be $O(N)$ for each test case.

PREREQUISITES:

Stack, Dynamic programming

PROBLEM:

Given an array of N integers $A_1, A_2, \ldots, A_N$, a function $f(i,j)$ is defined as follows:
$f(i,j) = A_i + max(A_i, A_{i+1}) + max(A_i, A_{i+1}, A_{i+2}) \cdots + max(A_i, A_{i+1}, \ldots, A_j)$.

We need to compute the value $\sum_{i=1}^{N} \sum_{j=i}^{N} f(i,j)$ modulo $10^9 + 7$.

QUICK EXPLANATION:

- Let $max_{i,j}$ be defined as $max(A_i, A_{i+1}, \ldots, A_j)$.
- The problem can be reformulated as: $\sum_{i=1}^{N} \sum_{j=i}^{N} max_{i,j} \cdot (N + 1 - j)$.
- Let us define a dp state where $dp_i = \sum_{j=i}^{N} max_{i,j} \cdot (N + 1 - j)$ for all $1 \leq i \leq N$.
- $dp_i = A_i \cdot \sum_{j=i}^{ind-1} (N + 1 - j) \; + \; dp_{ind}$ where ind is the nearest index greater than i where $dp_i \geq dp_{ind}$.

EXPLANATION:

Let $max_{i,j}$ be defined as $max(A_i, A_{i+1}, \ldots, A_j)$.

Now, there are a total of $N + 1 - j$ terms which have $max_{i,j}$ component inside them. They are $f(i,j), f(i,j+1), f(i,j+2) \ldots, f(i,N)$.

Therefore, our problem can be formulated as to find the value $\sum_{i=1}^{N} \sum_{j=i}^{N} g(i,j)$ where $g(i,j) = max_{i,j} \cdot (N + 1 - j)$.

Let us define a dp state where $dp_i = \sum_{j=i}^{N} g(i,j)$ for all $1 \leq i \leq N$.

Let us define ind as the **nearest index** greater than i for which $A_{ind} \geq A_i$. This means that for all k from $i+1$ to $ind - 1$, we have $max_{i,k} = A_i$. Also, for all k from ind to N, we have $max_{i,k} = max_{ind,k}$ since $A_{ind} \geq A_i$.

Therefore,

$$dp_i = \sum_{j=i}^{N} max_{i,j} \cdot (N + 1 - j)$$

$$\implies dp_i = \sum_{j=i}^{ind-1} A_i \cdot (N + 1 - j) \; + \; \sum_{j=ind}^{N} max_{ind,j} \cdot (N + 1 - j)$$

$$\implies dp_i = \sum_{j=i}^{ind-1} A_i \cdot (N + 1 - j) \; + \; dp_{ind}$$

$$\implies dp_i = A_i \cdot \sum_{j=i}^{ind-1} (N + 1 - j) \; + \; dp_{ind}$$

$\sum_{j=i}^{ind-1} (N + 1 - j)$ can be simplified easily if we know the value of $\sum_{j=i}^{ind-1} j$ which has the value equal to $\frac{ind \cdot (ind-1)}{2} - \frac{i \cdot (i-1)}{2}$.

The only thing left is to find the nearest index greater than i which has $A_{ind} \geq A_i$. This is a **classic problem** which can be done for all indices in $O(N)$ using a stack.

PROBLEM:

You wish to place K servers across N locations. The locations are arranged in a circular manner. You want to place the servers such that the maximum clockwise distance between any two adjacent servers is as less as possible. Find this minimum possible distance that can be achieved, and also the minimum number of pairs of servers separated by this distance.

EXPLANATION:

Let's visualize this problem as we have a circle of length N and we made K cuts on this circle. Then we divide the circle from where the cuts had been made. Hence we will get K pieces of the circle. We need to minimize the maximum length possible of any piece.

The optimal way to divide the circle into K pieces is to divide it into the pieces of length N/K. But after dividing it into K pieces there might be some residual that is still left. Hence we will have two cases:

CASE 1 : When $(N \bmod K = 0)$

- In this case when N is completely divisible by K, then there will be no residual left after dividing the circle into K pieces of length N/K. In this case the maximum length that is possible is N/K and the number of pieces of this length are K.

CASE 2: When $(N \bmod K \neq 0)$

- In this case, when N is not completely divisible by K then after dividing the circle into K pieces of length N/K, there will be a residual left of length $(N \bmod K)$.
- Since K is always greater than $(N \bmod K)$. Hence from the residual, we will give unit length to $(N \bmod K)$ pieces such that the length of $(N \bmod K)$ pieces will increase by a unit length.
- Hence, In this case the maximum length that is possible is $(N/K + 1)$ and the number of pieces of this length are $(N \bmod K)$.

TIME COMPLEXITY:

$O(1)$ per test case

PROBLEM:

Chef is working on his swap-based sorting algorithm for strings.

Given a string S of length N, he wants to know whether he can sort the string using his algorithm.

According to the algorithm, one can perform the following operation on string S any number of times:

- Choose some index i $(1 \leq i \leq N)$ and swap the i^{th} character from the front and the i^{th} character from the back.
  More formally, choose an index i and swap S_i and $S_{(N+1-i)}$.

For example, $\underline{\text{d}}\text{cb}\underline{\text{a}}$ can be converted to $\underline{\text{a}}\text{cb}\underline{\text{d}}$ using one operation where $i = 1$.

Help Chef find if it is possible to sort the string using **any** (possibly zero) number of operations.

EXPLANATION:

Let us first find out the string that the Chef will end up with, given that he performs the swap optimally.

Let us consider the index i and $j = (N + 1 - i)$. Also, let $i \leq j$. If $S_i \leq S_j$ then we do not need to swap these characters (as they are in already in correct order). Otherwise, we will swap S_i and S_j.

Let us first perform the above operation by iterating over the complete string, and let us denote the final string as S'. If S' is sorted, then our answer is **YES**, otherwise our answer will be **NO**.

To check if S' is sorted, we can iterate over the string and check that $S_i \leq S_{i+1}$ for all i such that $1 \leq i \leq N - 1$. Another way to check is to use the *sort* function which is usually present in the Standard Template Library (or similar libraries) in most of the languages.

TIME COMPLEXITY:

$O(N)$ or $O(N \cdot \log N)$ for each test case, depending on the implementation details.

PROBLEM:

Given arrays A and B of size N. You can swap A_i and A_{i+1} with cost $A_i - A_{i+1}$. Determine the minimum total cost to make A equal to B.

EXPLANATION:

Suppose we do a number of swaps to A to make it equal to B, and the element at index i in the original array is now at index pos_i.

I claim that the total cost incurred in the above case equals $\sum A_i(pos_i - i)$.

Proof

Let pos_i represent the current position of, the element at index i in the original array. Initially, $pos_i = i$ for all i.

Everytime we swap element A_i (of the original array) with the element A_j to its right:

- pos_i increases by 1,
- pos_j decreases by 1,
- the cost increases by $A_i - A_j \implies$ the cost increases by A_i and decreases by A_j.

We can therefore notice that everytime pos_i is increased by 1, the cost increases by A_i. Similarly, the cost decreases by A_i everytime pos_i is decreased by 1.

In the end, the total number of times the cost increases by A_i is equal to $pos_i - i$ (the cost decreases if $pos_i - i$ is negative).
Therefore, we can calculate the cost contributed by each A_i separately and then add them together, giving us the equation $\sum A_i(pos_i - i)$.

When the elements of A are distinct, array pos is unique and the total cost is therefore fixed. What about the case when A has repeating elements?

I claim that any sequence of swaps that makes A equal B has the same total cost. That is, say there exists i, j $(i \neq j)$ such that $A_i = A_j$. Then, the total cost to rearrange A such that A_i and A_j are at indices pos_j and pos_i (instead of pos_i and pos_j) is the same.
(The proof of this is trivial and left to the reader as an exercise).

Therefore, given A and B, it suffices to find any *valid* mapping pos such that $B_{pos_i} = A_i$. We can then use the equation given above to calculate the answer.

Implementation

Create array of pairs $C = \{(A_1, 1), (A_2, 2), \ldots, (A_N, N)\}$ and $D = \{(B_1, 1), (B_2, 2), \ldots, (B_N, N)\}$.

Sort the elements of C and D by the first value of their pairs, in ascending order (breaking ties arbitrarily). Then, since arrays A and B have the same elements, it is easy to see that the first value of C_i equals the first value of D_i for all i.

So, set the value of $pos_{C_i.second}$ as $D_i.second$, for all i. The validity of this mapping can be shown trivially.

TIME COMPLEXITY:

We sort the array of pairs C and D in $O(N \log N)$ each. We then iterate from 1 to N once, to generate the array pos and calculate the answer. The total time complexity is therefore

$$O(2 \times N \log N + N) \approx O(N \log N)$$

per test case.

PROBLEM:

Given $n \times m$ grid. A thief is trying to escape from a policeman. The thief is currently in cell (x, y) in the grid, while the policeman is in cell (a, b).

The thief needs to reach the cell (n, m) to escape. In one second, the thief can move either right, or down. The policeman can move right, or down, or (right + down) or stay in same cell in one second.

The policeman catches the thief if he's in the same cell as the thief at any point of time, and he had reached that cell strictly before the thief. Find if the thief shall escape, or not, if both of them move optimally.

QUICK EXPLANATION:

Just calculate minimum time for thief and policeman to reach the cell (n, m). Lets say policeman takes T_{police} and thief takes T_{thief} time to reach the cell (n, m). Now if $T_{police} < T_{thief}$, policeman will catch him.

EXPLANATION:

Thief is initially at the cell (x, y). It can move one step only in right or down in one second. Lets say it takes T_{thief} time to reach the cell (n, m) then
$T_{thief} = (n - x) + (m - y)$

Policeman is initially at the cell (a, b). It can move one step in right or down or (right + down) in one second. Lets say it takes T_{police} time to reach the cell (n, m) then
$T_{police} = max(n - a, m - b)$

Why

Because policeman will first move (right + down) as many steps as it can. Then left over steps either in right or down depending on the situation whether he ends up in last row or last column respectively.

Finally we can compare the times thief and policeman took to reach the cell (n, m). If $T_{police} < T_{thief}$ then the thief is caught.

Why we check for only last cell?

Lets say policeman is able to catch the thief at a cell (p, q). Now lets calculate the time taken for thief and policeman to reach the cell (n, m) from the cell (p, q).
$T_{thief} = (n - p) + (m - q)$
$T_{police} = max(n - p, m - q)$

Observe that $max(n - p, m - q) \leq (n - p) + (m - q)$ hence $T_{police} \leq T_{thief}$.

If policeman caught the thief at the cell (p, q) that means he had reached at the cell (p, q) before the thief. In this case, policeman will also be able to reach at the cell (n, m) before the thief because $T_{police} \leq T_{thief}$. Hence policeman will also be able to catch him at cell (n, m). Hence we can directly check for the cell (n, m) only.

TIME COMPLEXITY:

$O(1)$ per test case

PREREQUISITES:

Ceil function, Heaps

PROBLEM:

For the chef's exam, there are M subjects and N topics. The i^{th} topic belongs to $C_i{}^{th}$ subject and takes T_i hours to complete. There is K hours time in the test. We need to find the maximum marks possible for the chef to score in this exam if the score is calculated as follows: $\lceil \frac{x_1}{2} \rceil + \lceil \frac{x_2}{2} \rceil + \cdots + \lceil \frac{x_M}{2} \rceil$ if x_1 topics are chosen from subject 1, x_2 topics are chosen from subject 2, $\ldots x_M$ topics are chosen from subject M.

In this problem, $\lceil . \rceil$ denotes the ceil function.

QUICK EXPLANATION:

- This problem can be solved constructively. Initially we are are at state $x_1 = 0, x_2 = 0, \ldots x_M = 0$ and for each state we try to increase score by 1 to go to another state until the total time doesn't exceed K.
- Suppose we are at some state with values $x_1, x_2, \ldots x_M$. In order to increase score by 1, we need to add topics for the subject with minimum cost where $cost_i$ is defined as follows:
- If $x_i = 0$, $cost_i$ is defined as minimum time required to increase the number of topics of subject i by 1. Else x_i must always be odd and $cost_i$ is defined here as the minimum time to increase the number of topics of subject i by 2.
- We can proceed in this manner by utilizing a min heap to get the index i with minimum cost at each stage and keep on incrementing scores until there are no possible topics left we can add for a particular subject in order to increase the score by 1.

EXPLANATION:

The first observation we need is that if we select x_i topics from the i^{th} subject, to minimize time, we obviously need to select the first x_i values of the topics of subject i when they are sorted in ascending order by time.

The second observation is that x_i **is either** 0 **or an odd number** in the optimal answer.

Proof

Suppose there is some $x_i > 0$ and is even. From the properties of ceil function, we know that, $\lceil \frac{x_i}{2} \rceil = \lceil \frac{x_i - 1}{2} \rceil$ as x_i is even. Thus, removing an extra topic doesn't effect the score but will infact benefit us since we are lowering the time required for this same score. Therefore, we can always remove a topic and make x_i odd in this case.

Let us solve this problem constructively. Suppose we are at some **state** where already $x_1, x_2, x_3, \ldots x_M$ topics are selected from the corresponding subjects respectively. Now say **we want to increase the total score further by** 1**.** The main question under consideration is how can we go about it?

The main idea is that we calculate the minimum time $cost_i$ needed to increase the score by 1 if we add topics from subject i for each $1 \leq i \leq M$. Now we need to add topics to that subject which has value $min(cost_1, cost_2, \ldots cost_M)$ in order to increase the score by 1.

If we know how to calculate $cost_i$, our job is done since at every stage we can put $cost_1, cost_2 \ldots cost_M$ in a **min heap** and pop the items and perform the updates accordingly. (more details in code). This procedure is done as long as we do not cross the total time K.

Let us now calculate $cost_i$. Already x_i topics from this subject are selected and we want to increase the score by 1 by adding some more topics with minimum extra time added. This can be solved in two cases depending on $x_i = 0$ or an odd number.

- **Case 1:** $x_i = 0$
  In this case, we can simply add **one topic** belonging to subject i with minimum time in order to increase the score by 1. Therefore, $cost_i$ is the minimum time needed to increase the number of topics of subject i by 1.

- **Case 2:** x_i is odd.
  In this case, we need to add **atleast two topics** belonging to subject i in order to increase the score by 1. This is because of the property of ceil function, $\lceil \frac{x_i}{2} \rceil = \lceil \frac{x_i+1}{2} \rceil$ when x_i is odd. Therefore, $cost_i$ is the minimum time needed to increase the number of topics of subject i by 2.

Knowing all this, we initially start with the state $x_1 = 0, x_2 = 0, \ldots x_M = 0$ and slowly increase the score by 1 by choosing subjects and the topics inside them accordingly. We can use a **min heap** to keep track of our states at each stage.

TIME COMPLEXITY:

$O(N \cdot (\log N + \log M))$ for each testcase.

PREREQUISITES:

Expected Values, Combinatorics

PROBLEM:

Chef, who is a chef by profession as well, has created N pies (enumerated 1 through N) and has placed them in a row. Now he wants to taste all the pies, but by following a rather peculiar order of tasting.

At each step, he chooses (uniformly randomly and independently) either the current leftmost pie or the rightmost pie, and eats that pie.

For example, if $N = 7$, then one valid order of tasting the pies is $(1, 2, 7, 3, 4, 6, 5)$. However, $(1, 2, 7, 6, 4, 5, 3)$ is not a valid order.

Chef can't wait to taste all the pies, so for each i $(1 \leq i \leq N)$, he wants to know the expected value of the number of steps he'll need to eat the i^{th} pie.

It can be shown that each expected value can be expressed as a fraction $\frac{P}{Q}$. You should compute $P \cdot Q^{-1}$ modulo $10^9 + 7$, where Q^{-1} denotes the modular multiplicative inverse of Q under $10^9 + 7$

EXPLANATION:

Let's denote the operation of choosing leftmost pie by L, and choosing rightmost pie by R.

Let us consider the P^{th} pie from the left, and try to calculate the Expected value for this pie.
There are two ways to pick this pie - either using an L operation, or using an R operation.

Let us focus on the case when the P^{th} pie was chosen using an L operation.

A Natural, but incorrect(?) approach

Assume that this pie was chosen after $P + K$ operations. This means that number of L operations will be P, and the number of R operations will be K. Also, the last operation should be L.

Number of ways to choose pie after $P + K$ operations such that the last operation is L is same as Number of ways to arrange $\#(P - 1)$ Ls and $\#K$ Rs. Also, Because choosing L and R are equiprobable, the probability of each such arrangement is $(\frac{1}{2})^{P+K}$.

Therefore, the Probability of choosing P^{th} pie after $P + K$ operations such that the last operation was L = $\binom{P-1+K}{K} \cdot (\frac{1}{2})^{P+K}$.

And the Expected value can be calculated as:

$$E_P = \sum_{K=0}^{N-P} \binom{P-1+K}{K} \cdot (\frac{1}{2})^{P+K} \cdot (P + K)$$

Notice the constraints on K. The maximum number of R operations can be $N - P$, as we want the P^th pie to be chosen by an L operation.

The issue with the above summation is, it is difficult to calculate its value efficiently.
Do let us know in the comments if you have been able to calculate it efficiently!

A Magical, working approach

In this approach, we will continue choosing the pies even after we have chosen the P^{th} pie. This complete simulation can be represented as a string of length N having L and R as characters.

Let say that in the complete process, the R operations were done a total of K times.
Also, let say that the R operations were done a total of i times before choosing the P^{th} pie using L operation.

Because the P^{th} pie is chosen using L operation, the maximum value of K can be $N-P$.

Total number of strings such that the P^{th} pie is chosen using L operation will be

$$S_P = \sum_{k=0}^{N-P} \sum_{i=0}^{k} \binom{P-1+i}{i} \cdot \binom{N-(P+i)}{k-i} \cdot (P+i)$$

On Simplifying

Clubbing first and third term, we get

$$S_P = \sum_{k=0}^{N-P} \sum_{i=0}^{k} P \cdot \binom{P+i}{i} \cdot \binom{N-(P+i)}{k-i}$$

Let $T = \sum_{i=0}^{k} \binom{P+i}{i} \cdot \binom{N-(P+i)}{k-i}$

On further Simplifying

We know that

$$(1-x)^{-N} = \sum_{r=0}^{\infty} \binom{N-1+r}{r} x^r$$

Consider the following two binomial expansions:

$$(1-x)^{-(P+1)} = \sum_{r_1=0}^{\infty} \binom{P+r_1}{r_1} x^{r_1}$$

$$(1-x)^{-(N-P-K+1)} = \sum_{r_2=0}^{\infty} \binom{N-P-K+r_2}{r_2} x^{r_2}$$

The coefficient of x^K in $(1-x)^{-(P+1)} \cdot (1-x)^{-(N-P-K+1)}$ can be written as

$$\sum_{i=0}^{k} \binom{P+i}{i} \cdot \binom{N-(P+i)}{k-i}$$

which is same as T. Hence,

$$T = x^K : (1-x)^{-(P+1)} \cdot (1-x)^{-(N-P-K+1)}$$

$$T = x^K : (1-x)^{-(N-K)}$$

$$T = \binom{N+1}{k}$$

Substituting back in S_P, we have

$$S_P = P \cdot \sum_{k=0}^{N-P} \binom{N+1}{k}$$

Note that the sum S_P has currently accounted only for the case when the P^{th} element is chosen using an L operation.
To account for the case when P^{th} element is chosen using an R operation, we can observe that P^{th} element from the left is $(N-P+1)^{th}$ element from the right, and therefore the contribution coming from this case will be same as the contribution from $(N-P+1)^{th}$ element from the left (because L and R operations are symmetric).

Therefore, the final value of S_P will be:

$$S_P = (P \cdot \sum_{k=0}^{N-P} \binom{N+1}{k}) + ((N-P+1) \cdot \sum_{k=0}^{P-1} \binom{N+1}{k})$$

Each term can be calculated efficiently using the running sum.

Finally, each string is equiprobable with probability $(\frac{1}{2})^N$. Therefore, the expected value for the P^{th} pie can be represented as:

$E_P = (\frac{1}{2})^N \cdot S_P$

TIME COMPLEXITY:

The above algorithm can be implemented in $O(N \cdot \log N)$ as well as $O(N)$ depending on the pre-computation and implementation details.

PROBLEM:

There are three people, and each of them has an unbiased 6-sided die. The result of rolling a die will be a number between 1 and 6 (inclusive) with equal probability.

The three people throw their dice simultaneously. In this game, the third person wins only if his number is strictly greater than the sum of the other two numbers. Given that the first person rolls the value X and the second person rolls the value Y, what is the probability the third person will win?

EXPLANATION:

The third player will win the game if and only if his number is greater than the sum of the other two numbers. Let us try to find his chances of winning the game:

As the third person has an unbiased 6-sided die, the number of possible outcomes is 6. Now let's try to find out the favorable outcomes:

Say S denotes the sum of the value of X and Y. Now for winning, the third person should get the number greater than S in his die.

Case 1: $S \geq 6$

- In this case, the number of favorable outcomes for the third person is 0 as the dice don't contain the number greater than 6.

Case 2: $S < 6$

- In this case, the third person will win if we score greater than S. He can score anything between the range $[S+1, S+2, \ldots, 6]$. Hence the number of favorable outcomes in this case are:

$$fav = 6 - S$$

Now as we know the number of favorable outcomes as well as the possible outcomes, we can easily find out the probability as:

$$probability = \frac{favorable}{possible}$$

Finally, we will output this probability.

PREREQUISITES:

Sum-over-subsets DP

PROBLEM:

You are given an array $M = [M_1, M_2, \ldots, M_N]$. Count the number of ordered triples of distinct indices (i, j, k) such that $(M_i \oplus M_j)\ \&\ M_k = M_i \oplus (M_j\ \&\ M_k)$.

QUICK EXPlANATION

- Note that any triple that satisfies this condition must satisfy $A_i\ \&\ A_k = A_i$, i.e, A_i must be a submask of A_k.
- This is also a sufficient condition for the triple to be good.
- Upon fixing a k, the number of possible choices of i can be found using sum-over-subsets DP.
- j can be chosen arbitrarily once i and k are fixed, leaving it with $N - 2$ choices.

EXPLANATION:

A common trick with problems dealing with bitwise operations is to treat each bit independently, so let's do that here.
This means that each of M_i, M_j, M_k can be either 0 or 1.

- Suppose $M_i = 0$.
  Then, $(M_i \oplus M_j)\ \&\ M_k = M_j\ \&\ M_k$ and $M_i \oplus (M_j\ \&\ M_k) = M_j\ \&\ M_k$ so both expressions are already equal regardless of choice of M_j and M_k.
- Suppose $M_i = 1$.
  Then, if $M_k = 0$ we have $(M_i \oplus M_j)\ \&\ M_k = 0$ and $M_i \oplus (M_j\ \&\ M_k) = 1$ regardless of what M_j is, which means they will never be equal.
  Thus, we must have $M_k = 1$. In this case, it can be verified that both equations evaluate to $1 \oplus M_j$, so they're equal and once again, the value of M_j doesn't matter.

This tells us that what M_j is doesn't matter at all, while M_i can be 1 only if M_k is also 1.
Extending this condition to more bits tells us that a triple is good if and only if M_i is a submask of M_k.

Thus, we have the following algorithm to compute the number of triples:

- First, fix which index is chosen as k.
- Then, count the number of indices which can possibly be i - this is exactly the count of integers M_i such that M_i is a submask of M_k.
- Finally, j can be freely chosen to be any of the remaining $N - 2$ indices.

The first part is easy to do — simply iterate over every index of the array. The third part is also trivial, which leaves the second.

The second part essentially requires us to solve the following problem:
Let F_x denote the number of indices i such that $M_i = x$. We would like to compute

$$S_x = \sum_{y \subseteq x} F_y$$

where $y \subseteq x$ means that y is a submask of x.

This is a classical problem, which can be solved in $O(B2^B)$ using sum-over-subsets DP, where B is the number of bits.

In this case, the bound $M_i \leq 10^6$ gives us $B = 20$, because $2^{20} > 10^6$.
If you do not know what sum over subsets DP is, please go through **this codeforces blog** .

The final solution to the problem is then simply:

- Compute the array S using SOS DP.
- iterate over each index $1 \leq k \leq N$.
- Add $(S_{M_k} - 1) \cdot (N - 2)$ to the answer. We subtract 1 from S_{M_k} because M_k is itself a submask of M_k, and we can't choose $i = k$.

TIME COMPLEXITY:

$O(N + B \cdot 2^B)$ per test case, where $B = 20$ for this problem

PROBLEM:

There are three **distinct** points - A, B, C in the $X-Y$ plane. Initially, you are located at point A. You want to reach the point C satisfying the following conditions:

- You have to go through point B.
- You can move in any of the four axis-parallel directions ($+X$, $-X$, $+Y$, $-Y$ direction). However, you can make **at most** one turn in the path from A to C.

Determine if it is possible to reach the destination C satisfying the above conditions.

EXPLANATION:

Let us see when it is possible to reach poin C satisfying all the conditions:

Claim 1: Point B should lie between point A and point C

Proof

Let us prove this by contradiction. Suppose we have co-ordiantes of point B as (x_B, y_B) such that x_B doesn't lies bewteen x_A and x_C. Similarly y_B doesn't lies between y_A and y_C.

Now let's see what will happen for co-ordinate x_B :

First we start from co-ordinate x_A and can reach to co-ordinate x_B without any turn. But after reaching co-ordinate x_B we need to go to co-ordinate x_C, for that we need to change our direction by $180°$ which is not allowed. Hence we need 2 turns of $90°$ to reach to point x_C from x_B

Similarly we can see for the y co-ordinate. Hence if point B doesn't lies between point A and point C then it would pe never possible to reach point C satisfying all the conditions.

Now if the above condition is satisfied then there one more condition should be satisfied:

Claim 2: Atleast one of the co-ordinate of point B should be equal to the respective co-ordinate of point A or point C

Proof

Let us again prove this by contradiction. Suppose we have co-ordiantes of point B as (x_B, y_B) such that x_B is neither equal to x_A nor x_C. Similarly y_B is neither equal to y_A nor y_C.

- To reach from point A to point B we need exactly one turn of $90°$. As we can reach to co-ordinate x_B from co-ordinate x_A without any turn, but then we need to change our direction to reach to co-ordinate y_B.
- Similary we need exactly one turn of $90°$ to reach point C from point A.

Hence there should be atleast one of the co-ordinate of point B should be equal to the respective co-ordinate of point A or point C

If both the conditions are satisfied then only it is possible to reach the condition C satisfying the all conditions.

TIME COMPLEXITY:

$O(1)$ per test case

PREREQUISITES:

Observation, Queue, Binary Search

PROBLEM:

There are N people, numbered from 1 to N. They go to a cinema hall. Each of them buys a ticket, which has a number written on it. The number on the ticket of the i^{th} person is A_i.

There are infinite seats in the cinema hall. The seats are numbered sequentially starting from 1. All the N people stand in a queue to get their respective seats. Person 1 stands at the front of the queue, Person 2 stands in the second position of the queue, so on up to Person N who stands at the rear of the queue. They were given seats in this manner:

Let the number on the ticket of the person currently standing in front of the queue be X. If the X^{th} seat is empty, the person gets out of the queue and takes the X^{th} seat. Otherwise, the person goes to the rear of the queue, and the number on his ticket is incremented by one - that is, it becomes $X + 1$.

Print the seat number occupied by each of the N people.

EXPLANATION:

Let us try to do as the problem speaks:

Whenever we find a person which has the number X on his ticket and if the X^{th} seat is empty, then we can simply allot a seat to that person and remove him from the queue. Otherwise, we increment his seat number and make him go to the last of the queue. We keep on doing so until there is no person left in the queue.

The method is correct but it is quite slow and hence it will result in TLE. One of the worst-case for this method is when all people standing in the queue want the same seat. Then it is quite easy to see that it takes a whole iteration of the queue to allot a seat to a single person.

Let’s see what are the optimizations we can do. Instead of finding a seat for a person, let’s try to find a person for the seat.

What is the largest seat number that will be allocated to any person standing in the queue?

- From the problem constraints $A_i \leq N$, it means initially no person can have a ticket that has a seat number greater than N. Now in the worst case, all the people have a ticket which has seat number N written on it.
- It is quite easy to see that in one whole iteration of the queue, at least one person will get a seat. Hence we can say that the largest seat that will be allocated to any person will be $2 * N - 1$.

In other words, we only need seat numbers 1 to $2 * N - 1$ to allocate seats to every person standing in the queue.

We can make another observation that a seat number S can only be allotted to those people which have a ticket number less than or equal to S.

Claim: A seat number S is allotted to a person who has a ticket number next smaller to S.

Proof

Suppose there are three people which have a ticket number lower than S. Let’s say their ticket numbers are a, b, and c such that $a < b < c$. Now in every iteration, each person ticket number will be increased by 1, and c being closest to S requires fewer iterations to get incremented to S. Once it reaches the value S, the person gets the seat S.

Also if there are two or more person which has same ticket number and is next smaller to S, then the person which is present earlier in the queue gets the seat S.

Now, once we know which seat will be allotted to which person, we can simply print the seat numbers occupied by every person.

Rest is implementation, try to implement it and if there are any doubts in implementation please let me know in the comments section.

TIME COMPLEXITY:

$O(N * log(N))$ per test case

PROBLEM:

You are given an array A of length N.

You have to partition the elements of the array into some **subsequences** such that:

- Each element A_i $(1 \leq i \leq N)$ belongs to **exactly one** subsequence.
- The **mean** of the mean of subsequences is **maximised**.

Formally, let $S_1, S_2, \ldots, S_K$ denote K subsequences of array A such that each element A_i $(1 \leq i \leq N)$ belongs to exactly one subsequence S_j $(1 \leq j \leq K)$.
Let X_j $(1 \leq j \leq K)$ denote the mean of the elements of subsequence S_j. You need to maximise the value $\frac{\sum_{j=1}^{K} X_j}{K}$.

Print the maximum value. The answer is considered correct if the relative error is less than 10^{-6}.

EXPLANATION:

Sort the array A in non-increasing order.
For a fixed $n_1 \leq n_2 \leq ... \leq n_k$, the sizes of the blocks in the partition, It is optimal to partition it into

$$[A_1, ...A_{n_1}], [A_{n_1+1}...A_{n_1+n_2}]....$$

Now fix some k, Consider $n_1 \leq n_2 \leq ... \leq n_k$, the sequence of partition sizes corresponding to optimum. If there are multiple such sequences, choose one with minimum n_1. Let $m_1, m_2, ...m_k$ be the means of corresponding partitions.
Assume $n_1 > 1$, We have

$$m_1 = \frac{(A_1 + ... + A_{n_1})}{n_1}$$
$$m_2 = \frac{(A_{n_1+1} + ...A_{n_1+n_2})}{n_2}$$

If we move A_{n_1} to second block, the new mean of first block is

$$m_{1'} = \frac{(A_1 + ...A_{n_1-1})}{(n_1 - 1)}$$

which is clearly $\geq m1$ as removing the minimum in a sequence increases it's mean and the new mean of second block is

$$m_{2'} = \frac{(A_{n_1} + A_{n_1+1} + ...A_{n_1+n_2})}{(n_2 + 1)}$$

Since A is in non-increasing order, $A_{n1} >= A_{n_1+i}$ for all $1 <= i <= n_2$ so $m_{2'}$ is clearly $\geq m_2$ because $A_{n_1} >= m_2$ so by shifting A_{n_1} to second block may give a better partition but not worse.
But we chose the partition with minimal n_1. A contradiction. So, $n_1 = 1$.
Using induction, we can prove that $n_i = 1$ for all $i < k$ and $n_k = n - k + 1$ is the optimal way to partition.

Thus we can choose values of k from 1 to N and calculate the corresponding mean, the maximum of which would be our answer.

PROBLEM:

Chef is given a contract to build towers in Chefland which are made by stacking blocks one above the other. Initially, there is only 1 block in the inventory and no tower has been built. Chef follows the following 2 steps in each operation:

- Step 1: Either build a new tower or update an existing tower that has been built in previous operations using the blocks currently present in the inventory. After this step, the size of the inventory reduces by the number of blocks used.
- Step 2: Suppose the tower Chef updated or built in Step 1 has B blocks after the step. Chef gets to add B new blocks to the inventory as a reward.

Find the maximum number of towers of height X Chef can build in M operations.

QUICK EXPLANATION:

- If $M < (ceil(log_2(X)) + 1)$, the answer is 0.
- Else, we can build $M - ceil(log_2(X))$ towers of height X.

EXPLANATION:

Observation

Note that the reward for an operation is equal to the number of blocks present in the last built or updated tower. To maximise this reward, it is optimal to update an existing tower instead of building a new one, since the former one would always have more blocks. If we keep updating the current tower, the reward increases exponentially.

Building a single tower of height X

Let us build the first tower of height X. We have 1 block initially. We use it to build a tower of height 1. We now get a reward of 1 block. If we update the current tower using this, the height of the tower becomes 2. After the reward for this move, the inventory size is also 2.

We continue using the whole inventory to build and update a single tower until it reaches the height X.

The inventory size forms a sequence while we do this process: $1, 1, 2, 4, 8, 16, 32, \ldots$.

Once the sum of this sequence reaches X, we have our first tower of height greater than equal to X.

Minimum operations required to reach the sum X can either be calculated using brute force (in $O(log_2 X)$ complexity) or using the formula $(ceil(log_2(X)) + 1)$ (in $O(1)$ complexity).

If the minimum operations required is greater than M, we cannot build even a single tower and the answer is 0.

Building maximum number of towers

Once we have built our first tower of required size, we have atleast X blocks in our inventory (due to our last operation). For all the remaining operations, we build a new tower of height X in each operation.

To sum up, the answer is 0, if $M < (ceil(log_2(X)) + 1)$, else, we can build $M - ceil(log_2(X))$ towers of height X in M operations.

TIME COMPLEXITY:

The time complexity is $O(1)$ per test case.

PREREQUISITES:

You should be familiar with DP and optimisation with Bitset

PROBLEM:

Chef has **three** storage containers with capacities X, Y and Z respectively. Each container can store **multiple** flavors of ice cream. Since Chef does not know which flavors Chefina would order, he wants to store **maximum number** of flavors in the storage containers.

Chef has a total of N flavors. In order to store the i^{th} flavor, Chef must select **exactly one** container and fill it with C_i amount of flavor i.

Note that the total amount of ice cream filled in a container should not exceed its capacity.

Find the **maximum number** of flavors Chef can store in the containers.

QUICK EXPLANATION:

- Observe that if r flavours are to be selected, selecting the minimum possible r elements would be optimal.
- Create a three dimensional $dp[i][j][k]$, where it is boolean denoting wheter it is possible to store i flavors with container of capacity j, k, Z. We report the first i, for which $dp[i][j][k]$ is zero $\forall\, j, k$ (0 based indexing).
- We optimise memory by observing dp[i] is only dependent on dp[i-1] and optimise time complexity using bitset.

EXPLANATION:

We first observe that if r elements can be stored, the first r flavors in the ascending sorted order of capacity can also be stored. This can be easily proven using contradiction. This follows the conclusion that r flavors can only be stored if and only if the least r flavors can be stored.

- Therefore, sort the flavors according to their capacity

Now we create a dp solution for our problem. Let $dp[i][j][k][l]$ denote a boolean if the least i elements can be stored using containers with capacity j, k, l.

- Flavor i can be stored in one of the three containers
- It follows the recursive relation that

$$dp[i][j][k][l] = dp[i-1][j-C_i][k][l] \vee dp[i-1][j][k-C_i][l] \vee dp[i-1][j][k][l-C_i]$$

We optimise the above dp solution using the fact that the sum of first i elements must be $\leq (j+k+l)$. Therefore, keeping two dimensions is enough.

- Let sum of first i flavors be denoted by
- The new recursive relation followed is

$$dp[i][j][k] = (dp[i-1][j][k] \wedge (S_i \leq (j+k+Z))) \vee dp[i-1][j-C_i][k] \vee dp[i-1][j][k-C_i]$$

This solution is still too slow to pass, we use bitset to optimise it further .
We create a bitset of length Y for each i and j.

- let e_i denote the first part ($S_i \leq (j+k+Z)$) we get $\max(w_i = S_i - j - Z, 0)$, making first w_i elements as zero we get $e_i = (dp[i-1][j] >> (w_i)) << w_i)$
- the second part is simply $dp[i-1][j-C_i]$

- the third part translates to $dp[i-1][j] << C_i$
- performing the or operation on these three gives us $dp[i]$

As $dp[i]$ is only dependent on $dp[i-1]$ we can optimise space using this fact

TIME COMPLEXITY:

$O(N \cdot X \cdot Y/64)$ for each testcase.

PREREQUISITES:

Greedy, Priority Queue

PROBLEM:

There are N training plans. Training plan i has effectiveness A_i, but requires that atleast B_i other training plans have been performed before selecting this plan.

If training plans $p_1, p_2, \ldots, p_k$ are performed, the score is equal to $\frac{\sum A_{p_i}}{k}$. Determine the maximum possible score attainable.

EXPLANATION:

We proceed greedily.

Let S represent the set of all (previously unselected) training plans that we can select currently. Initially, S consists of all plans i where $B_i = 0$.
Also, let set K comprise all plans that we have selected. Initially, K is empty. Finally, let ans represent the maximum score we have attained so far. Initially, $ans = 0$.

In each move, do the following:

- select the plan in S with the largest effectiveness value; break if S is empty.
- insert this plan into K and remove it from S.
- update $ans := \max(ans, \frac{\sum A_{K_i}}{|K|})$.
- insert all plans i with $B_i = |K|$ to S.

It is easy to see why this works. We pick plans one by one (till we can select no more). By the greedy criteria, it is optimal to always select a *selectable* plan with the highest effectiveness value. Finally, we update the best answer with the current score.

TIME COMPLEXITY:

Since each training plan is inserted/removed from the priority queue atmost once, the time complexity is therefore

$$O(N \log N)$$

per test case.

PREREQUISITES:

You should be familiar with the concept of shortest paths in a weighted graph. Specifically, this problem requires **Dijkstra's Algorithm**

PROBLEM:

You are given a graph with N vertices (numbered 1 to N) and M bidirectional edges, which doesn't contain multiple edges or self-loops — that is, the given graph is a simple undirected graph.

For each pair of vertices a, b such that $1 \leq a, b \leq N$, it is possible to add a new edge between vertices a and b to the graph, with a cost of $(a-b)^2$.

Find the minimum cost of adding edges so that vertex N is reachable from vertex 1.

QUICK EXPLANATION:

- Let us assume that in the optimal solution, we have added an edge (u, v). Let $u < v$ and $v - u \geq 2$. We can show that instead of adding this edge, we could have added $(u, u+1), (u+1, u+2) \ldots (v-1, v)$. This would have costed $v - u$.
- Consider the edge $(i, i+1)$. If it is already present in the graph, then it's cost is 0, otherwise it's cost is 1. Cost of all other edges which are already present in the graph is 0.
- In the above graph, we want to find out the length of shortest path from 1 to N.

EXPLANATION:

First of all, let us analyze our cost function. One of the most important observation to solve this problem is to observe that if we want to add an edge (u, v), such that $u < v$ and $v - u \geq 2$. We can instead add the edges $(u, u+1), (u+1, u+2) \ldots (v-1, v)$. This would result in a lower cost. Also, the cost of adding any one of this edge would be 1.

Now, our problem reduces to finding out the minimum number of edges of the form $(i, i+1)$, that we need to add in our graph, so that we can reach from 1 to N.

Let us add the edges $(i, i+1)$ in the above graph if they were not already present. Also, let us assign the weight 0 to all the edges that were already present in the graph, and 1 to the edges that we have just added. We can see that our problem is exactly same as finding the length of shortest path in the newly formed graph.

To find the shortest path, we can use **Dijkstra's Algorithm** or **0-1 BFS**.

TIME COMPLEXITY:

$O(M \cdot \log N)$ or $O(N + M)$, depending on the implementation.

PROBLEM:

Given a rooted tree of N nodes. Determine the sum of distance between nodes x and y over all tuples (u, v, x, y) such that

- $u < v$,
- u is the ancestor of x, v is the ancestor of y,
- neither u or v are ancestors of each other.

EXPLANATION:

The problem can be equivalently restated as:

Given a rooted tree of N nodes. Define $cost(x, y)$ as:

$$dist(x, y) * dist(x, p) * dist(y, p)$$

where p is the lowest common ancestor of x, y.
Determine the sum of $cost(x, y)$ over all ordered pairs (x, y).

Start by expanding the above equation to:

$$(dist(x, p) + dist(p, y)) * dist(x, p) * dist(y, p)$$
$$= dist(x, p)^2 * dist(y, p) + dist(y, p)^2 * dist(x, p)$$

Since the above equation is dependent on p in each term, we can try finding the answer for each p, rather than each ordered pair (x, y).

Let X_p be the set of direct children of node p. Then, let $C_{p,i}$ be the set of nodes in the subtree of node $X_{p,i}$ (including node $X_{p,i}$) for all valid i.

Slight mathematical definition

$\sum dist(S, p)$ represents the sum of distances between p and every node in set S.
$\sum dist(S, p)^2$ represents the sum of the square of distances between p and every node in set S.

The above equation, applied to node p is:

$$\sum_{i<j} (\sum dist(C_{p,i}, p)^2 * \sum dist(C_{p,j}, p) + \sum dist(C_{p,j}, p)^2 * \sum dist(C_{p,i}, p))$$

Bingo! We now have a formula for the answer that doesn't depend on the LCA function. How is that very helpful? The simplification gives the problem an elegant tree DP solution.

We jump to implementation. Let's calculate the answer, by calculating the value of the above expression for each p and adding them together.

Define $l_{X_{p,i}}$ and $l2_{X_{p,i}}$ as $\sum dist(C_{p,i}, p)$ and $\sum dist(C_{p,i}, p)^2$ respectively. Both arrays can be computed in $O(N)$ using tree DP.

How?

I leave the proof's to the reader, as they are trivial to deduce.

Define sz_u as the number of nodes in the subtree of node x (including itself).
The recurrence relations are then:

$$l_u = \sum(l_{X_u} + sz_{X_u})$$

and

$$l2_u = \sum(l2_{X_u} + 2 * l_{X_u} + sz_{X_u})$$

which can be computed recursively, using traditional tree DP.

(Start with deriving the first equation, which is done trivially. To then derive the second equation, use the same logic and apply the formula $(a+1)^2 = a^2 + 2 * a + 1$)

The above equation then becomes:

$$\sum_{i<j}(l2_{X_{p,i}} * l_{X_{p,j}} + l2_{X_{p,j}} * l_{X_{p,i}})$$

which can be calculated in linear time using a modified form of sliding window.

How?

Create two temp variables t and $t2$, both initially 0.

Iterate over X_p; let the current node be $X_{p,i}$. At each step, add $t2 * l_{X_{p,i}} + t * l2_{X_{p,i}}$ to the answer. Then, add $l_{X_{p,i}}$ and $l2_{X_{p,i}}$ to t and $t2$ respectively.

It can be shown easily, that this method correctly calculates the above summation in linear time.

Summing the last equation for all p gives us the desired answer !

TIME COMPLEXITY:

Computing array's l and $l2$ takes $O(N)$ time. Computing the final equation above, for each p, takes $O(|X_p|)$.
The total complexity is thus:

$$O(N + N + |X_1| + |X_2| + \dots) = O(N + N + N) \approx O(N)$$

PREREQUISITES:

Depth first search, trees, dynamic programming (dp)

PROBLEM:

Given a rooted tree (at vertex 1) with N vertices with each vertex i having value A_i, two players Alice and Bob play a game. The game is played alternatively with Alice starting first with a pointer at vertex 1. In each turn, the player (who is currently at vertex i) can move the pointer to a child vertex of i and adds the value of that child vertex to that score. The game ends when a player is unable to make a move. Alice wants to minimize the score and Bob wants to maximize the score. Now we need to find out what would be the final score at the end if **each player** can skip their move **atmost K times**.

QUICK EXPLANATION:

- The final answer is always independent of K because if one player makes a skip in order to favour them, the other player can always make a skip right away in order to avoid such scenario.
- Thus, let us solve for the modified problem where there is no option to skip a move.
- Now the problem can be solved by 2D dp. Let player 0 be Alice and player 1 be Bob. $dp_{i,j}$ be the total score obtained when player j starts the game with the pointer at vertex i.
- $dp_{i,0}$ = $\min_{\forall c \in children(i)}(dp_{c,1} + A_c)$
- $dp_{i,1}$ = $\max_{\forall c \in children(i)}(dp_{c,0} + A_c)$
- The final answer will be $dp_{1,0}$.

EXPLANATION:

First let us solve the problem with having $K = 0$ i.e, no player can skip their moves. Let us assume player 0 is Alice and player 1 is Bob. Let us also assume that $children(i)$ is the list consisting of every child of veretx i.

The key idea to solve this is by using dynamic programming. We need to define a state. Let $dp_{i,j}$ denote the final score obtained if the game starts with player j and the pointer is at vertex i. Here $1 \leq i \leq N$ and $0 \leq j \leq 1$.

The base case will be as follows: When the vertex i is a leaf, $dp_{i,0} = dp_{i,1} = 0$.

If the vertex i is not a leaf, we can perform the transitions as follows:

- $dp_{i,0}$ = $\min_{\forall c \in children(i)}(dp_{c,1} + A_c)$. This is because person 0 wants to minimize the score and therefore we find the minimum over all the children that person can move the pointer to.
- $dp_{i,1}$ = $\max_{\forall c \in children(i)}(dp_{c,0} + A_c)$. This is because person 1 wants to maximize the score and therefore we find the maximum over all the children that person can move the pointer to.

Doing dfs on tree will help us compute all the dp values. $dp_{1,0}$ will be the answer for this case.

Now let us come back to our original problem: What happens if $K > 0$? I claim that even if $K > 0$, our final answer is the same as the case with $K = 0$. We can argue about this in the following way:

Suppose Alice need to make a move when the pointer is at vertex i. Assume no skips by any player happened till now. Now consider a scenario where Alice makes a skip that reduces the total score. I claim that this scenario cannot happen as Bob can always make a skip right away in order to avoid such scenario.

Similarly, we can argue that Bob cannot make the first skip.

Therefore, the final answer is $dp_{1,0}$ which is independent of K.

TIME COMPLEXITY:

$O(N)$ for each testcase.

PREREQUISITES:

Euler totient function

PROBLEM:

Given a positive integer N, we need to find the number of tuples (A, B, C, D) where $1 \leq A, B, C, D \leq N$ and $A \cdot B = C \cdot D$.

QUICK EXPLANATION:

- This problem is equivalent to finding the number of tuples (A, B, C, D) where $1 \leq A, B, C, D \leq N$ and $\frac{A}{B} = \frac{C}{D}$.
- The number of tuples (A, B, C, D) for which $A = B$ and $\frac{A}{B} = \frac{C}{D}$ will be N^2.
- If we find the number of tuples (A, B, C, D) for which $A \geq B$, (let this be x), we can also find the final answer which will be $2 \cdot x - N^2$.
- Every fraction $\frac{A}{B}$ can be reduced into a unique irreducible fraction $\frac{X}{Y}$ i.e, $GCD(X, Y) = 1$.
- In order to solve the case for $X \geq Y$, we can iterate over every X from 1 to N and add $\phi(X) \cdot \lfloor \frac{N}{X} \rfloor \cdot \lfloor \frac{N}{X} \rfloor$ to the answer where ϕ is the euler totient function. This is because the number of Y for which $GCD(X, Y) = 1$ is $\phi(X)$ and each of the irreducible fraction $\frac{X}{Y}$ contributes $\lfloor \frac{N}{X} \rfloor \cdot \lfloor \frac{N}{X} \rfloor$ fraction pairs to the answer.

EXPLANATION:

Let us consider the equation $A \cdot B = C \cdot D$. This equation can be rewritten as follows: $\frac{A}{C} = \frac{D}{B}$. Therefore, we can reformulate the problem as follows: Given a positive integer N, we need to find the number of tuples (A, B, C, D) where $1 \leq A, B, C, D \leq N$ and $\frac{A}{B} = \frac{C}{D}$. The answer for this problem must be the same as the answer for our original problem.

The number of fractions $\frac{A}{B}$ for which $A = B$ will be N. They are $\frac{1}{1}, \frac{2}{2}, \frac{3}{3}, \ldots \frac{N}{N}$. Therefore, the number of tuples (A, B, C, D) where $A = B$ and $\frac{A}{B} = \frac{C}{D}$ will be the number of pairs we can form with these N fractions which is $N \cdot N$.

Let us solve the case for $A \geq B$. Let this be x. Then, by symmetry, the number of tuples (A, B, C, D) for $A \leq B$ will also be x. Therefore, our final answer will then be
$x + x -$ (number of tuples (A, B, C, D) where $A = B$ and $\frac{A}{B} = \frac{C}{D}$)
$= x + x - (N \cdot N)$
$= 2 \cdot x - (N \cdot N)$.

Therefore, we could easily find the final answer if we have the answer for the case $A \geq B$.

The most crucial property of $\frac{A}{B} = \frac{C}{D}$ is that both $\frac{A}{B}$ and $\frac{C}{D}$ can be reduced to a unique irreducible fraction $\frac{X}{Y}$ i.e, $GCD(X, Y) = 1$.

Thus if we go through every **irreducible fraction** $\frac{X}{Y}$ where $X \geq Y$ and calculate the number of tuples (A, B, C, D) where $\frac{A}{B} = \frac{C}{D} = \frac{X}{Y}$, then we are done.

The number of fractions which are equal to $\frac{X}{Y}$ is $\lfloor \frac{N}{X} \rfloor$. They are $\frac{X}{Y}, \frac{2 \cdot X}{2 \cdot Y}, \frac{3 \cdot X}{3 \cdot Y}, \ldots \frac{\lfloor \frac{N}{X} \rfloor \cdot X}{\lfloor \frac{N}{X} \rfloor \cdot Y}$.

The number of pairs we can make from these fractions are $\lfloor \frac{N}{X} \rfloor \cdot \lfloor \frac{N}{X} \rfloor$.

Let us fix some X, and let *tot* be the number of possible Y where $X \geq Y$ and $gcd(X, Y) = 1$. Clearly, $tot = \phi(X)$ where ϕ is the **euler-totient function**. We can precompute the values of $\phi(X)$ for all $1 \leq X \leq N$.

The number of irreducible fractions $\frac{X}{Y}$ where $X \geq Y$ is $\phi(X)$ and each of them contribute $\lfloor \frac{N}{X} \rfloor \cdot \lfloor \frac{N}{X} \rfloor$ to the answer.

From this information we can calculate the answer for the case $A \geq B$ as follows:
Iterate over X from 1 to N and add $\phi(X) \cdot \lfloor \frac{N}{X} \rfloor \cdot \lfloor \frac{N}{X} \rfloor$ to the answer.

By calculating the answer for $A \geq B$, we can easily find the total number of tuples (A, B, C, D) for which $\frac{A}{B} = \frac{C}{D}$ as explained above.

TIME COMPLEXITY:

$O(N \log N)$ or $O(N \log \log N)$ depending on the implementation for precomputation of ϕ values.

PREREQUISITES:

Greedy

PROBLEM:

You are given two arrays A and B with distinct elements, and an integer C. You may perform the following operations - select an element in either array that is $\leq C$ and remove it. Then, increase the values of all elements in that array by 1, and decrease the values of all elements in the other array by 1.

Determine if it's possible to empty both arrays.

EXPLANATION:

Firstly, observe that if a merchant has x items priced $\leq C$, all x items can be purchased by buying from the greatest to smallest priced item.

How?

Let $c_1 < c_2 < \cdots < c_x \leq C$ be the costs of the items.
Now, we can buy items from c_x to c_1, and it is easy to see that the cost of the items (increasing by 1 after each move) never exceeds C.

Without loss of generality, let's say a valid solution $[a_1, b_1, a_2, b_2, \dots]$ exists, where we buy a_1 items from A, followed by b_1 items from B, followed by a_2 items from A and so on (the case where we buy first from B can be handled similarly).

Claim: It is possible to simplify the above solution to $[p, q, r]$, that is, we buy p items from A, followed by all q $(q = M)$ items from B and finally the remaining r $(r = N - p)$ items from A.

Proof

We show that any sequence of the form $[a_1, b_1, a_2, b_2]$ can be done in ≤ 3 moves. This can then be applied recursively to any valid solution, to reduce it to a 3 move one.

Case 1: $a_1 \geq b_1$
Here, it is easy to see that $[a_1 + a2, b_1 + b_2]$ is a valid sequence of moves, that buys the same number of items from both merchants.

Why?

Since $a_1 \geq b_1$, the price of all items in A after move 2 is more than the initial price. Thus, its trivial to deduce that the a_2 items purchased in move 3 could have be bought in move 1 itself, proving the validity of our simplified sequence.

Case 2: $a_1 < b_1$
A bit of casework shows that $[0, b_1 - a_1, a_1 + a_2, (b1 - a_1) + b_2] \implies [b_1 - a_1, a_1 + a_2, (b1 - a_1) + b_2]$ is an equivalent sequence of 3 moves, where we start by buying from B instead.

Why?

Since $a_1 < b_1$, it is clear that there are atleast $b_1 - a_1$ items in B originally, with cost $\leq C$.
The cost of the items in A at the end of the first move (in our simplified sequence) decreases by $b_1 - a_1$, which is the same amount it decreases by at the end of the second move in the original sequence.
Thus, it is possible to buy $a_1 + a_2$ items from A in move 2, proving the validity of our new sequence.

With this claim, the solution is straightforward - A valid solution exists only if we can buy $\max(0, B_M - C)$ elements from A (which is the minimum number of buys we need to be able to buy everything from B at once), and after buying everything from B, we can buy all remaining items in A (for this, the price of all items in A should be $\leq C$).

Here's an equivalent mathematical expression of the above.
Let p equal the number of items in A with price $\leq C$. Also, let $q = \max(0, B_M - C)$ Then, a valid solution exists iff:

$$(q \leq p) \text{ and } (A_N + q - M \leq C) \text{ holds.}$$

Don't forget to check similarly for the case where we first buy from B!

TIME COMPLEXITY:

Since we iterate over the arrays exactly once, the time complexity is

$$O(N + M)$$

per test case.

PROBLEM:

Chef has two binary strings A and B, each of length N. He can perform the following operation on A any number of times:

- Choose L and R $(1 \leq L \leq R \leq N)$, such that, in the **substring** $A[L, R]$, the number of $1s$ is **not equal** to the number of $0s$ and **reverse** the substring $A[L, R]$.

Find whether Chef can convert the string A into the string B by performing the above operation **any** (possibly zero) number of times on A.

QUICK EXPLANATION:

- If A and B are not anagrams, then the answer is **NO**
- If any one of A or B has only alternate $0's$ and $1's$ (like *101010* or *010101*), then the answer is **YES** iff $A = B$, otherwise answer is **NO**
- Otherwise, there exists an index i such that $A_i = A_{i+1}$. Similarly, there exists an index j such that $B_j = B_{j+1}$.
- We can show that if there is an index i such that $A_i = A_{i+1}$, we can sort the string A using the above operations. Similarly, we can sort B, and because the operations are reversible, we can show that A can be converted to B (by first sorting A, and then following the operations used in sorting B in reverse order).

EXPLANATION:

Because the mentioned operation cannot change the number of $0's$ or $1's$ in the string, the answer will always be **NO** if A and B are not anagrams.

Now let's say that for every $i (1 \leq i < N), A_i \neq A_{i+1}$. In other words, A has alternate $0's$ and $1's$. In this case, note that every even length substring has equal number of $0's$ and $1's$, and therefore we can only reverse substrings of odd lengths. However, every odd length substring is a palindrome, and hence reversing it doesn't modify A. In simpler words, we can never modify A using the above operations. So, the answer will be **YES** only if $A = B$, otherwise the answer will be **NO**.
Note that the operations are reversible, and converting A to B is equivalent to converting B to A. Hence, if B has alternate $0's$ and $1's$, then the above argument holds.

Now we have an index i such that $A_i = A_{i+1}$. Similarly, we have an index j such that $B_j = B_{j+1}$. We will show that if there is such an index, then we can sort our string. If this is true, we can first sort A. Let S be the ordered set of operations which are applied to B while sorting it. Because of the reversible nature of the operation, we can apply the operations of S in the reverse order to string A, and get the string B. Note that this is true because A and B are anagrams.

Sorting string A when $A_i = A_{i+1}$
Without loss of generality, let $A_i = 0$

Bringing all 1's which are present on the left side of the pair to the right side

Let's see this through an example. We will use 0-based indexing.
Let $A = 1010001$. Let us consider the pair $\{A_4, A_5\}$. We will start iterating towards left until we reach the first *1*. In this case, the substring would be *1000*, where the last two *0's* represents our pair. We will always have unequal number of *0's* and *1's* as there is single *1*, while at least two *0s*.

We can reverse this substring, bringing the *1* towards the right side (and effectively shifting our pair towards left). We can continue this process to bring all the *1s* towards right.

Bringing all 0's which are present on the right side of the pair to the left side

We will start iterating towards right until we reach the first 0 at k^{th} index. So, our substring is $A_iA_{i+1} \dots A_k$, with $A_i, A_{i+1}, A_k = 0$.
If $A_{i+2} \dots A_k$ is valid, then we can directly reverse this substring. If not, then both $A_iA_{i+1} \dots A_k$ and $A_{i+1} \dots A_k$ are both valid. So, we can have $A_iA_{i+1} \dots A_k \rightarrow A_kA_{k-1} \dots A_{i+1}A_i \rightarrow A_kA_iA_{i+1} \dots A_{k-1}$, and hence effectively bringing A_k to the left side.

If we have $A_i = 1$, then we essentially need to do the same thing. Hence we have shown that if there are two consecutive equal characters, we can sort our string, and therefore, we can convert A to B as explained above.

TIME COMPLEXITY:

$O(N)$ or $O(N \cdot \log N)$ for each test case, depending on the implementation.

PROBLEM:

Alice and Bob play a game on an array of N integers. They alternate moves, with Alice making the first move.

The rules are as follows:

1. On **their first move**, a player can pick **any** element in the array, add its value to their score, and then **remove** it from the array.
2. On a move that is **not their first move**, the player should pick an element with the **opposite** parity of the element chosen on **their previous move**, add its value to their score, and then **remove** it from the array.
3. If a player cannot make a move, either because the array is empty or it doesn't contain an element of the parity they need, the player skips their turn.
4. The game ends when both players are unable to move.

Note that the parity of an element chosen by a player depends only on the parity of their own previously chosen element — it does not depend on what their opponent last chose.

Determine the optimal score of Alice if both players play optimally, each attempting to **maximize** their own score. If there are multiple ways to obtain maximum score for themselves, the players will adopt a strategy that will **maximize** the score of their opponent whilst obtaining their maximum score themselves.

Note that it is not necessary to use all the elements in the array.

EXPLANATION:

This game can be played in four ways:

- Both $Alice$ and Bob picks an even number.
- Both $Alice$ and Bob picks an odd number.
- $Alice$ picks an even number while Bob picks an odd number.
- $Alice$ picks an odd number while Bob picks an even number.

In either of the four ways, the game would then proceed the same for both, i.e picking of alternating even-odd numbers in decreasing order.

Since Alice makes the first move, so she would select either odd or $even$ depending upon which would maximise her final score. For Alice's selection, Bob will then have two options to either select $even$ or odd depending upon which would maximise his final score.
Assuming a function called $f(mask)$ which gives the pair of score of $Alice$ and Bob with respect to the given $mask$, whose first bit represents the choice of Bob and second bit represents the choice of Alice. Here $mask$ is define as follows:
$mask = 0$ => Both Alice and Bob picks $even$ number
$mask = 1$ => Alice picks an $even$ number while Bob picks an odd number.
$mask = 2$ => $Alice$ picks an odd number while Bob picks an $even$ number.
$mask = 3$ => Both Alice and Bob picks odd number.

Now for a particular choice of $Alice$, her final score would depend on the choice of $Bob's$ choice who aims to maximise his own score.

- Alice choses even initially, then her score would be
  - $f(0)$ if Bob score's maximises if he also choses even initially
  - $f(1)$ if Bob score's maximises if he choses odd initially
  - $max(f(0), f(1))$ if Bob's score is the same in both of his choices.
- Alice choses odd initially, then her score would be
  - $f(2)$ if Bob score's maximises if he choses even initially
  - $f(3)$ if Bob score's maximises if he also choses odd initially.
  - $max(f(2), f(3))$ if Bob's score is the same in both of his choices.

Out of these two cases of $Alice's$ choice, her score would be the maximum out of the two.

TIME COMPLEXITY:

$O(NlogN)$ for each test case

PREREQUISITES:

Modular Multiplicative Inverse, Bitwise xor

PROBLEM:

We are given an array A of N integers $A_1, A_2, \ldots, A_N$ and an array B of M integers $B_1, B_2, \ldots, B_M$. We need to find the total number of ordered pairs (i, j) for which the following conditions hold good:

- $1 \leq i \leq N$
- $1 \leq j \leq M$
- P divides $(A_i \cdot (A_i \oplus B_j) - 1)$
- $(A_i \oplus B_j) < P$

EXPLANATION:

- The key idea of this problem is to iterate over i from 1 to N and find the number of possible j for which the given conditions hold true.
- Suppose we fix some i for which $A_i \mod P = 0$. Then, according to the third condition, $-1 \mod P = 0$ which is impossible since $P \geq 2$. Thus, we cannot find any possible j in this case.
- Let us fix some i for which $A_i \mod P \neq 0$. Now, from the third condition, we have
$$(A_i \cdot (A_i \oplus B_j) - 1) \mod P = 0$$
$$\implies (A_i \cdot (A_i \oplus B_j)) \mod P = 1$$
$$\implies (A_i \oplus B_j) \mod P = A_i^{-1} \mod P$$
$$\implies (A_i \oplus B_j) = A_i^{-1} \mod P \text{ (according to the last condition)}$$
- Since P is a prime, the modular inverse always exists for A_i where $A_i \mod P \neq 0$ and it can be computed with the help of Fermat's little theorem. Finally, we get the value of B_j as $B_j = (A_i \oplus (A_i^{-1} \mod P))$. We can initially precompute this count of B_j and add it to the answer.

TIME COMPLEXITY:

$O(N \log P)$ for each test case for calculating modular inverse. If we precompute the counts using a map instead of hash map, we get the time complexity as $O(N \log N + N \log P)$.

PROBLEM

Given a binary string S and an integer C, you are allowed to perform circular on S any number of times. Does there exist any rotation such that every pair of adjacent ones is separated by at most C zeros.

QUICK EXPLANATION

Assuming the last character to be adjacent to first, we can find the number of zeros between each pair of adjacent ones in a list. Now, the rotation of binary string is equivalent to deleting at most one element of this list. So if rest of the elements are up to C, then the answer is YES.

EXPLANATION

For this problem we'll consider an example first. Assume S = `010001010`. Now we shall consider all rotations of this string and compute the number of zeros between each adjacent pair of ones.

- `010001010`: The number of zeros between adjacent ones are $\{3, 1\}$
- `001000101`: The number of zeros between adjacent ones are $\{3, 1\}$
- `100100010`: The number of zeros between adjacent ones are $\{2, 3\}$
- `010010001`: The number of zeros between adjacent ones are $\{2, 3\}$
- `101001000`: The number of zeros between adjacent ones are $\{1, 2\}$
- `010100100`: The number of zeros between adjacent ones are $\{1, 2\}$
- `001010010`: The number of zeros between adjacent ones are $\{1, 2\}$
- `000101001`: The number of zeros between adjacent ones are $\{1, 2\}$
- `100010100`: The number of zeros between adjacent ones are $\{3, 1\}$

We can see that the list of number of zeros between adjacent ones changes only when some `1` moves from last position to first position. But why does that happen?

It's because the last and second last occurrence of `1` is no longer adjacent, and additionally, last occurrence of one becomes the first occurrence now, so it is adjacent to second occurrence in rotated string, hence number of zeros between them are added.

Simple idea, Non-simple implementation

This way, we can actually maintain the queue type structure, and simulate the whole process, where queue supports the following operations

- Add element at beginning
- Remove element from end
- Find maximum of elements in queue

In order to support last operation, there are various methods, like mentioned **here** or **implementing queue using two stacks** and using **minimum stack** approach.

Can we observe more?

Above approaches were correct, although require too much implementation for an easy problem like this, so let's observe a bit more.

We saw that list of number of zeros between each pair of adjacent ones got affected only when some `1` moves from last to first. Let's assume for a moment that last and first character of S are adjacent.

Considering S = `010001010`, now the list of number of zeros between adjacent ones is $\{3, 1, 2\}$. It is easy to see that all rotations of S have this same list of number of zeros between adjacent ones.

Hence, rotation only affect the number of zeros between last and first occurrence of one, effectively removing it from the list.

So, we can see that if we compute the list of zeros between adjacent ones when S is assumed to be cyclic, we can delete any one element from the list. It can be seen that the new list after deletion shall correspond to a cyclic rotation of original string S.

Simpler solution, simpler life

So now we have the list of number of zeros between adjacent ones when S is assumed to be cyclic, we know we can delete exactly one element. And we want the remaining elements in list to be up to C. Hence, it is optimal to delete the largest element from the list and check if remaining elements are up to C or not.

It is equivalent to checking whether the second-maximum element of this list is up to C or not. This can all be done with much simpler implementation.

Genera Note: There are many such problems where easy idea leads to complicated implementation, but with some more insights, solutions can become a lot more simpler.

TIME COMPLEXITY

The time complexity is $O(N)$ per test case.
The space complexity is $O(N)$ per test case.

PROBLEM:

Given and array A of N integers, the weirdness of the subarray $A_L, A_{L+1}, \ldots, A_R$ denoted by $w(L, R)$ is defined as the number of indices $L \leq i \leq R$ for which A_i equals the number of occurrences of A_i in that subarray. We need to find the value of $\sum_{L=1}^{N} \sum_{R=L}^{N} w(L, R)$.

EXPLANATION:

- First let us create an array $vals_x$ for every unique value x present in A. $vals_x$ denotes the array of integers $i_1, i_2, i_3, \ldots i_{cnt_x}$ for which $A_{i_1} = A_{i_2} = \cdots = A_{i_{cnt_x}} = x$. Here, cnt_x denotes the total number of values in A equal to x.
- Suppose we fix some x. Let us consider a technique of sliding window with window of size x and keep sliding the window on the array $vals_x$.
- Suppose our window is like $i_j, i_{j+1}, \ldots, i_{j+x-1}$. Now let us solve this problem which is, **how much does this window contribute to the answer?**
- To solve this, we need to find the number of subarrays which contain this window and have no other index k apart from those in window with $A_k = x$.
- Clearly, the starting point of such subarray can be in the range $[i_{j-1} + 1, i_j]$ and the ending point of such subarray can be in the range $[i_{j+x-1}, i_{j+x} - 1]$. (If i_{j-1} doesn't exist, we can replace it with 0 and if i_{j+x} doesn't exist, we can replace it with $N + 1$).
- Therefore, the total number of such subarrays for the current window are $(i_j - i_{j-1}) \cdot (i_{j+x} - i_{j+x-1})$.
- Each subarray have x values equal to x, so each subarray contributes x to the answer. So, the total contribution will be $x \cdot (i_j - i_{j-1}) \cdot (i_{j+x} - i_{j+x-1})$.
- We can similarly do this for every window of every unique value x in the array A and add its contribution to the answer.

TIME COMPLEXITY:

$O(N)$ for each testcase.

PREREQUISITES:

Dynamic Programming.

PROBLEM:

There are K travellers who want to travel through a country one after the other. The country is in the form of a 2-d grid and is divided into N rows and M columns, such that each cell of the grid can be considered to be a city.

Each city is initially marked with either 0 or 1. Let city (i,j) (the city at the intersection of the i_{th} row and j_{th} column) be marked with number C_{ij}. If a traveller is in the city (i,j), then

- If C_{ij}=0 and $j < M$, then the traveller moves to city $(i,j+1)$ and at the same time C_{ij} is changed to 1.
- If C_{ij}=1 and $i < N$, then the traveller moves to city $(i+1,j)$ and at the same time C_{ij} is changed to 1.

If the traveller cannot make a move from a city (i,j), then this city is considered to be the destination of that traveller. Each traveller starts their journey from city $(1,1)$, and all but the first traveller start their journey only once the previous one has reached their destination. Find the destination city of the K_{th} traveller.

Quick Explanation

We will find the state of the grid when k_{th} traveller starts its journey. Now, for any cell (excluding the ones on right and bottom boundary), its value in the required state is only dependent number of travellers that passed through this cell. So, initially we will compute for each cell ,the number of traveller that reached here. This can be computed using dp and initial state of cells.

For cells on right boundary, if its initial state is 0, then it will always remain 0. Otherwise if it's 1, the first traveller that reaches this city will cross it and move to next, after that its value changes to 0 and it remains same thereafter. Similarly for bottom row.

Once we find the final state of the grid, we can move the k_{th} traveller according to the rules of the question to get to its final position.

Explanation

Let us call $dp[i][j][l]$ as the number of travellers visiting the state at the grid (i,j) when its C_{ij} value is l. The first part of the problem requires calculation of this dp. Let us proceed with that.

- First covering the base case:

$$dp[0][0][0] = \begin{cases} k/2 \text{ if } k \text{ is even} \\ k/2+1 \text{ if } k \text{ is odd and } grid[0][0] = 0 \end{cases}$$

$$dp[0][0][1] = \begin{cases} k/2 \text{ if } k \text{ is even} \\ k/2+1 \text{ if } k \text{ is odd and } grid[0][0] = 1 \end{cases}$$

For rest cases assuming, we take another variable $total$ as:

$$total = dp[i-1][j][0] + dp[i][j-1][1]$$

- Case when covering the last column, i.e $j = m-1$. Here if the C_{ij} value of a cell is 0 then it remains 0, otherwise it would change to 0 as soon as any visiter visits this cell and then it remains 0

$$dp[i][j][0] = \begin{cases} total \\ total - 1 \ if\ total > 0\ and\ initial\ value\ of\ cell\ is\ 1 \end{cases}$$

$$dp[i][j][1] = \begin{cases} 0 \\ 1 \ if\ total > 0\ and\ initial\ value\ of\ cell\ is\ 1 \end{cases}$$

- Case when covering the last row, i.e $i = n - 1$. Similar analysis here as that for last column.

$$dp[i][j][1] = \begin{cases} total \\ total - 1 \ if\ total > 0\ and\ initial\ value\ of\ cell\ is\ 0 \end{cases}$$

$$dp[i][j][1] = \begin{cases} 0 \\ 1 \ if\ total > 0\ and\ initial\ value\ of\ cell\ is\ 1 \end{cases}$$

- Rest all cases:

$$dp[i][j][0] = \begin{cases} \frac{total}{2} \\ \frac{total}{2} + 1 \ if\ k\ is\ odd\ and\ initial\ value\ of\ cell\ is\ 0 \end{cases}$$

$$dp[i][j][1] = \begin{cases} \frac{total}{2} \\ \frac{total}{2} + 1 \ if\ k\ is\ odd\ and\ initial\ value\ of\ cell\ is\ 1 \end{cases}$$

With that we computed our dp. This would help in determining the final state of the grid when the k_{th} traveller will start travelling. This is because by knowing the number of people visiting a cell, we can determine the number of times the value of that cell has changed and thus find the final state of the cell when k_{th} traveller is visiting the cell. Thus knowing the state of the cell, we can just move according to the rules given in the question to get to the final position of the k_{th} traveller.

TIME COMPLEXITY:

In this algorithm, we iterated through the 2D array which takes $O(NM)$ time. Also while traversing the 2D array according to the rules of the question for the final state of the grid before the k_{th} traveller starts its journey, at worst we would reach the bottom right cell that would again take $O(NM)$ time. Thus in total the time complexity is $O(NM)$.

PREREQUISITES

Dynamic Programming, Bitmasking

PROBLEM

You are given an array $A = [A_1, A_2, \ldots, A_N]$ containing N **distinct** integers. Count the number of ways to form (unordered) sets of disjoint increasing **subsequences** of A.

Formally, count the number of sets $S = \{S_1, S_2, \ldots, S_k\}$ such that:

- Each S_i is an increasing subsequence of A.
- If $i \neq j$, S_i and S_j are disjoint, i.e, $i \neq j \implies S_i \cap S_j = \emptyset$

Note that it is *not necessary* that the sequences $S_1, S_2, \ldots, S_k$ form a partition of A - in other words, some elements of A may not be in any chosen subsequence.

Two sets are considered equal if they contain the same subsequences. For example, the sets $\{[1,2],[3]\}$ and $\{[3],[1,2]\}$ are considered to be the same and should only be counted once.

Note that the final answer can be rather large, so compute its remainder after dividing it by $10^9 + 7$.

QUICK EXPLANATION

- Let us process the elements from left to right one by one, and try to maintain active sequences.
- Use dynamic programming using bitsets to maintain active sequences, we can either not include the current element in any sequence, create a new sequence with the current element, or add the current element at the end of any active sequence.
- Set bit in bitmask would represent an active sequence ending at that position.

EXPLANATION

Since the order of sequences does not matter, we will process the elements from left to right, and try adding elements either at the end of one of the active lists, or create a new list starting with this element, or not add this element at all.

Considering example $A = [1,2,3,4]$, Suppose we have processed first three elements. We may have $\{[1,2],[3]\}$, or $\{[2,3]\}$ or so on. What information do we need?

We need the last elements of all active sequences. For $\{[1,2],[3]\}$, we can either add 4 at end of $[1,2]$, or at end of $[3]$ or start a new sequence $[4]$, or not add this element at all.

If, instead of $\{[1,2],[3]\}$, we had $\{[2],[3]\}$, the treatment would have been the same. We don't care about elements of sequences other than the last element, because whether or not we can append at the end, solely depends on the last element.

Hence, we can represent $\{[1,2],[3]\}$ by set $\{2,3\}$ denoting the endpoints of active sequences. Similar set for sequences $\{[2,3]\}$ would be $\{3\}$.

Programatically, this set can be represented by a bitmask, where ith bit set would imply there's a sequence ending at A_i.

We consider elements from left to right and try to update these.

Let's assume $f_x(mask)$ denote the number of unordered set of increasing sequences considering first x elements of A. We aim to compute $f_x(mask)$ from $f_{x-1}(mask)$ for any $mask$.

Initially, we start with $f_0(0) = 1$, the empty sequence.

The transitions are as follows:

- If we do not include x-th element at all, then it contributes $f_{x-1}(mask)$ to $f_x(mask)$
- If we start a new sequence with x-th element, then it contributes $f_{x-1}(mask)$ to $f_x(mask + 2^x)$
- If for some active sequence ending at element $y < x$ such that $A_y < A_x$, A_x can be added at end of such sequence. It contributes $f_{i-1}(mask)$ to $f_x(mask - 2^y + 2^x)$

$f_x(mask)$ can be represented by a 2D array and the DP table can be built iteratively or recursively as well.

In the end, the final answer would be the sum of $f_N(mask)$ for all masks in $[0, 2^N - 1]$.

For recursive implementation, see setter's solution. For iterative implementation, refer to Editorialist's solution.

TIME COMPLEXITY

The time complexity would be $O(N * 2^N)$ per test case.

The actual number of operations can be estimated as $\sum_{i=1}^{N} i * 2^i$.