Problem A. Delete, Deduct, and Destroy

Problem Setter: Pritom Kundu
Tester: Rafid Bin Mostofa
Alt Writer: Irfanur Rahman Rafio
Category: Data Structures, Math

At first let's try to understand the properties of $f(x, k)$. For that, let's take a small example and see what's going on.
Let x be a three digit number where the hundreds digit is $a(1 \le a \le 9)$, the tens digit is $b(0 \le b \le 9)$ and the units digit is $c(0 \le c \le 9)$. So, we can write $x = 100a + 10b + c$.

For $k = 3$, we have to delete the digit a. So, $y = 10b+c$ and $f(x, 3) = x-y = (100a+10b+c)-(10b+c) = 100a$.
For $k = 2$, we have to delete the digit b. So, $y = 10a+c$ and $f(x, 2) = (100a+10b+c)-(10a+c) = 90a+10b$.
For $k = 1$, we have to delete the digit c. So, $y = 10a+b$ and $f(x, 1) = (100a+10b+c)-(10a+b) = 90a+9b+c$.

Let $h(z, k)$ be the number of number of x satisfying $f(x, k) = z$.
Assuming z is a three digit number, we can see that $h(z, 3)$ is the number of ways to choose x, which is to say, choose (a, b, c) such that $100a = z$. This means that if z is not divisible by 100, then $h(z, 3) = 0$. Otherwise, $h(z, 3) = 100$ (1 way to choose a and 10 ways each to choose b and c).
Similarly, $h(z, 2)$ is the number of ways to choose (a, b, c) such that $90a + 10b = z$. Here, if z is not divisible by 10, then $h(z, 2) = 0$. Otherwise, we can use $a = \left\lfloor \frac{z}{90} \right\rfloor$ and $b = \frac{z - 90a}{10}$ and there are 10 ways to choose c. So, $h(z, 2) = 10$.
Finally, $h(z, 1)$ is the number of ways to choose (a, b, c) such that $90a+9b+c = z$. Here, we can use $a = \left\lfloor \frac{z}{90} \right\rfloor$, $b = \left\lfloor \frac{z - 90a}{9} \right\rfloor$ and $c = z - 90a - 9b$. So, $h(z, 1) = 1$.

From the above example, we can see that for a 3 digit number x:

- If $k = 3$, then $h(z, k) = 100$ if z is divisible by 100, otherwise 0.
- If $k = 2$, then $h(z, k) = 10$ if z is divisible by 10, otherwise 0.
- If $k = 1$, then $h(z, k) = 1$.

Since $g(z)$ is the sum of all $h(z, k)$ for all valid k, we can say that for a 3 digit number z: $g(z) = h(z, 1) + h(z, 2) + h(z, 3)$, which can be calculated as follows:

- If z is divisible by 100, then $g(z) = 100 + 10 + 1 = 111$.
- Else if z is divisible by 10, then $g(z) = 10 + 1 = 11$.
- Else, $g(z) = 1$.

This generalizes for any number of digits. For an n digit number z, we can say that if p is the largest integer such that z is divisible by 10^p, then:

$$g(z) = \sum_{k=0}^{p} 10^k = \frac{10^{p+1} - 1}{9}.$$

However, in our calculation, we made an assumption that when z is not divisible by 10, then $h(z, 1) = 1$ because there is only one way to choose (a, b, c). But, if $c = 0$, then we can add 9 to c and subtract 1 from b to get another valid choice of (a, b, c).
Similarly, if z is divisible by 10 but not by 100, then if $b = 0$, we can add 9 to b and subtract 1 from a to get another valid choice of (a, b, c).
Finally, if z is divisible by 100, then if $a = 9$, we can subtract 9 from a and add 1 to the digit before a (which

is not present in our example) to get another valid choice of (a, b, c).
This can be generalized for any number of digits that if z is divisible by 9, then we can double the value of $g(z)$ that we calculated before because we can always find another valid choice of digits by borrowing from the next non-zero digit.

Now, we need to make sure that x has at least two digits and no leading zeros. If we consider x to be a two digit number, then $x = 10b + c$, where $1 \leq b \leq 9$ and $0 \leq c \leq 9$. The possible values of $f(x, k)$ will be:

- For $k = 2$, $f(x, 2) = 9b + c$.
- For $k = 1$, $f(x, 1) = 10b$.

If $b > 0$, then all values of $f(x, k)$ will be greater than or equal to 9. So, if $z < 9$, then $g(z) = 0$. Also, when $z = 9$, we have $b = 1$ and $c = 0$, which gives us $f(x, 2) = 9$. $b = 0$ and $c = 9$ is not a valid choice since it leads to a leading zero in x. So, $g(9) = 1$.

Now, we can summarize our findings as follows:

- If $z < 9$, then $g(z) = 0$.
- If $z = 9$, then $g(z) = 1$.
- Else, let p be the largest integer such that z is divisible by 10^p.
  - If z is not divisible by 9, then $g(z) = \dfrac{10^{p+1} - 1}{9}$.
  - If z is divisible by 9, then $g(z) = 2 \times \dfrac{10^{p+1} - 1}{9}$.

The formula above allows us to calculate $g(z)$ directly for any given z (using modular multiplicative inverse or prefix sum). The only information we need are the value of p, whether z is divisible by 9 or not, and whether z is less than, greater than or equal to 9. For answering queries, we need to keep track of these three pieces of information efficiently.

To check the value of p, we only need to know the number of trailing zeros in z. This can be achieved by maintaining a set of positions where the digit is non-zero, or by maintaining a segment tree or a binary indexed tree to quickly find the rightmost non-zero digit.
To check whether z is divisible by 9, we can maintain the sum of the digits of z modulo 9. Whenever a digit is updated, we can update this sum accordingly.
Finally, to check whether z is less than or equal to 9, we only need to check the most significant digit of z. If it is in the rightmost position and is less than 9, then $z < 9$. If it is equal to 9, then $z = 9$. If the most significant digit is not in the rightmost position, then $z > 9$.

Problem B. Your Next Line Is, "What A Cool Problem!"

Problem Setter: Irfanur Rahman Rafio
Tester: Hasinur Rahman, Sharif Minhazul Islam Emon
Alt Writer: Arnob Sarker
Category: Ad-hoc

Hint 1: Try to find an optimal strategy for Caesar.
Hint 2: Analyze the samples and try to understand what's going on.
Hint 3.1: How does the vocabulary {`king`, `ping`, `ring`, `sing`, `wing`} guarantee Joseph's victory in the third test case?
Hint 3.2: Can you find a better vocabulary which will allow Joseph to use the same strategy?
Hint 3.3: Try this vocabulary: {`kkkk`, `pppp`, `rrrr`, `ssss`, `wwww`}.

Solution:

Let's analyze the problem case by case.

If $n \geq a$, Caesar can simply try all letters in the alphabet. This guarantees that he will discover the hidden word regardless of Joseph's strategy. Therefore, in this case, the answer is `"NO"`.

If $n \geq v$, Caesar will first assume that the first word of the vocabulary is the hidden word. If Joseph's response marks it as incorrect, he can decrease the vocabulary size by 1. Since Caesar can afford $(v-1)$ incorrect attempts, he can eliminate all words one by one until there is only one left. Thus, in this case, the answer is also `"NO"`.

If $n < a$ and $n < v$, Joseph can choose the vocabulary such that it contains words that have the same letter in all l positions. For example, if $l = 3$ and the alphabet is {`a`, `b`, `c`, `d`}, Joseph can choose the vocabulary {`aaa`, `bbb`, `ccc`} if $v = 3$. [If $v > a$, Joseph can simply put random words in the last $v - n$ positions of the vocabulary which he will not use.] When Caesar makes a guess, Joseph will delete all words containing that letter from the vocabulary. Since $n < v$, after n wrong attempts, there will still be at least one word left in the vocabulary. Joseph can then reveal that word as the hidden word, proving that all his responses were correct. Thus, in this case, Joseph can guarantee victory, and the answer is `"YES"`.

So, the final solution is simply `"YES"` if $n < a$ and $n < v$, and `"NO"` otherwise.

Problem C. Least Compatible Ancestor

Problem Setter: Shahjalal Shohag
Tester: Irfanur Rahman Rafio
Alt Writer: Irfanur Rahman Rafio, Rafid Bin Mostafa
Category: Trees, Dynamic Programming, Bitmasks

Key Observation

The condition $a_{\text{lca}(i,j)} \neq \text{lca}(a_i, a_j)$ for all distinct pairs (i, j) can be reformulated into a simpler equivalent condition:

If node u is an ancestor of node v in the tree, then a_u cannot be an ancestor of a_v.

Proof of Equivalence

Necessity: Suppose u is a proper ancestor of v (i.e., $u \prec v$). Then $\text{lca}(u, v) = u$. If a_u is an ancestor of (or equal to) a_v, then $\text{lca}(a_u, a_v) = a_u$. This gives us:

$$a_{\text{lca}(u,v)} = a_u = \text{lca}(a_u, a_v)$$

which violates the original condition. Therefore, if $u \prec v$, then a_u cannot be an ancestor of a_v.

Sufficiency: Consider any two distinct nodes i and j, and let $v = \text{lca}(i, j)$. Then v is an ancestor of both i and j. By our equivalent condition, a_v is not an ancestor of a_i and a_v is not an ancestor of a_j.

However, $\text{lca}(a_i, a_j)$ must be an ancestor of both a_i and a_j. If $\text{lca}(a_i, a_j) = a_v$, then a_v would be an ancestor of both a_i and a_j, contradicting our condition. Therefore:

$$a_{\text{lca}(i,j)} = a_v \neq \text{lca}(a_i, a_j)$$

Thus, the conditions are equivalent.

Solution Approach

We use dynamic programming with bitmasks. For each node u in the tree, we precompute subtree[v] — a bitmask representing all nodes in the subtree rooted at v.

Define $dp[u][\text{mask}]$ as the number of ways to assign values to nodes in the subtree of u such that:

- Values in the bitmask mask cannot be used (they are forbidden).
- The assignment satisfies the ancestor constraint within this subtree.

For each node u, we iterate over all possible values val that can be assigned to u (values not in mask). When we assign val to node u:

- All nodes in subtree[val] become forbidden for descendants of u (since u is their ancestor).
- We compute the DP for all children of u with the updated forbidden mask mask | subtree[val].
- The number of ways is the product of ways for all children.

The final answer is computed as $dp[1][0]$ since the tree is rooted at node 1.

Time Complexity: $O(n^2 \cdot 2^n)$

Space Complexity: $O(n \cdot 2^n)$.

Problem D. Magical Flower Garden

Problem Setter: Tariq Hasan Rizu
Tester: Rafid Bin Mostofa
Alt Writer: Irfanur Rahman Rafio, Rafid Bin Mostofa
Category: Range Query, Bitmasks

Key Insight

Let $B_i = \text{popcount}(A_i)$ denote the saturation of the i-th flower. The score of each subarray is determined by the *maximum* saturation inside it. For every position i, let us compute the total contribution of B_i as the *maximum* in all subarrays where i is the dominating position.

Initial Computation

Let, L_i is the index of the previous greater or equal element of i and R_i is the index of the next greater element of i. Then i is the unique maximum for exactly $(i - L_i)\,(R_i - i)$ subarrays. Thus,

$$\text{beauty}(A) = \sum_{i=1}^{n} (i - L_i)(R_i - i)\, B_i.$$

We can compute L_i and R_i in $O(n)$ using monotonic stacks and store each element's individual contribution for updates.

Type 2 Query

We can maintain a running sum of contributions to answer the Type 2 queries in $O(1)$ time.

Type 1 Query

We can maintain 31 priority queues, one for each possible saturation, allowing us to locate the target flower in $O(\log n)$. Since A_i changes only by OR-ing with q, its saturation can increase by at most 31 bits.

For each bit that turns from 0 to 1, the saturation of i increases by 1. We can remove i's old contribution and then locally adjust the monotonic links- if i now overtakes its next-greater neighbour, we fix that neighbour's contribution and update its pointer. Likewise, all consecutive previous elements with the same saturation are updated so that their next-greater pointer becomes i.

Time Complexity

$O(31(n+d)\log n)$

Problem E. The Perfect View

Problem Setter: Muhiminul Islam Osim
Tester: Shahjalal Shohag
Alt Writer: Irfanur Rahman Rafio
Category: Geometry

Simplifying the Problem

The first and most crucial step is to reduce the infinite 2D plane to a finite set of candidate locations. The "scenic value" of a point (the number of triangles it's inside) only changes when it crosses a triangle's edge. The N cafe locations are vertices for all $N \cdot M(M-1)/2$ triangles, making them the most critical points.

We can hypothesize that an optimal point Q will always be found **infinitesimally close** to one of the N cafe locations C_i.

This insight transforms the problem into N independent subproblems. For each cafe C_i, we must find the maximum number of triangles $\Delta C_i L_j L_k$ that a point Q (placed very close to C_i) can be strictly inside.

Finding the Best Direction

For a fixed cafe C and a point Q infinitesimally close to it, the condition "strictly inside $\Delta C L_j L_k$" simplifies:

1. **The Base:** The line $L_j - L_k$ is "far away" from C. Q will always be on the same side of this line as C, so this condition is automatically satisfied.
2. **The Cone:** Q must be in the angular cone formed at vertex C by the rays $C - L_j$ and $C - L_k$.
3. **The Constraint:** The problem's constraints imply this angular cone must be **strictly less than 180 degrees**. A cone of 180° or more is invalid.

Our subproblem is now: **For a given cafe C, find a single direction (a ray) that is contained within the maximum number of valid ($< 180^\circ$) cones $\Delta C L_j L_k$.**

Rotational Sweep-Line

This problem is a perfect fit for a **rotational sweep-line algorithm**. The score of a ray only changes when it crosses one of the M landmark rays, so we only need to test the M angular sectors they create. A sweep-line algorithm does this efficiently.

For a given cafe C, we first translate all landmarks so C is at the origin. Then, we find the polar angle of each landmark and then sort them. This takes $O(M \log M)$ time.

The algorithm then proceeds in two steps:

1. Initial Score Calculation ($O(M)$) We must find the score for a starting ray (e.g., just before the first landmark). This score is the number of cones ΔCL_jL_k that "wrap around" the 0/360 degree mark (i.e., where the counter-clockwise angle from L_j to L_k is $> 180°$). A naive $O(M^2)$ check can be optimized to $O(M)$ using a **two-pointer scan**. We iterate with a pointer j and maintain a second pointer i to count how many i's satisfy `angles[i]` $<$ `angles[j]` $- \pi$.

2. The Sweep and Update ($O(M)$) Once we have the `initial_score`, we simulate a ray sweeping 360°, "crossing" each landmark L_i. When our ray crosses L_i, the score changes based only on the $M - 1$ cones that have L_i as a vertex:

- **Cones Entered:** We *enter* every cone ΔCL_iL_j where L_j is in the 180° arc "ahead" of L_i.
- **Cones Left:** We *leave* every cone ΔCL_kL_i where L_k is in the 180° arc "behind" of L_i.

The total change in score is:

$$\text{change} = (\text{num\_landmarks\_ahead}) - (\text{num\_landmarks\_behind})$$

This update is $O(1)$ amortized time by using another "fast" pointer (`p_opposite`) to track the landmark 180° away. As i moves forward, `p_opposite` also only moves forward.

Complexity Analysis

- **Per cafe:** $O(M \log M)$ (sort) + $O(M)$ (initial score) + $O(M)$ (sweep) = $O(M \log M)$
- **Total Time Per Test Case:** $O(N \cdot M \log M)$
- **Total Space:** $O(M)$ (to store angles per cafe)

Problem F. Over Counting

Problem Setter: Yeamin Kaiser
Tester: Irfanur Rahman Rafio
Alt Writer: Irfanur Rahman Rafio
Category: Combinatorics

Let a be a binary string of length n, and $c = f(a)$. By definition, c_i is the number of indices $j < i$ such that $a_j > a_i$.

- If $a_i = 1$, the condition $a_j > a_i$ becomes $a_j > 1$, which is impossible in a binary string. Thus, if $a_i = 1$, then $c_i = 0$.
- If $a_i = 0$, the condition $a_j > a_i$ becomes $a_j > 0$, which means $a_j = 1$. Thus, if $a_i = 0$, then c_i is the number of 1s at positions $j < i$.

The value $g(a)$ is the total number of **inversions** in the array c.

We need to find the pairs (i, k) such that $1 \le i < k \le n$ and $c_i > c_k$. Consider the possible values of a_i and a_k:

1. **If $a_i = 1$:** Then $c_i = 0$. Since $c_k \ge 0$, $c_i > c_k$ is impossible.
2. **If $a_k = 0$:** Then c_k is the count of 1s before k.

- **If $a_i = 0$ (and $i < k$):** Then c_i is the count of 1s before i. Since $i < k$, $c_k \geq c_i$. The condition $c_i > c_k$ is impossible.

3. **If $a_i = 0$ and $a_k = 1$:** Then $c_k = 0$. The condition becomes $c_i > 0$. From the definition of c, $c_i > 0$ if and only if $a_i = 0$ and there exists at least one '1' at some position $p < i$.

Combining these, an inversion (i, k) exists in c if and only if $i < k$, $a_i = 0$, $a_k = 1$, and there is at least one '1' at a position $p < i$.

Let p_0 be the index of the *first* '1' in the string a. Then $g(a)$ counts all pairs (i, k) such that $p_0 < i < k$, $a_i = 0$, and $a_k = 1$. This is equivalent to counting the number of triples (p, j, k) such that $p < j < k$, $a_p = 1$, $a_j = 0$, $a_k = 1$, and p is the index of the **first '1'** in the string.

We need to compute $\sum g(a)$ over all distinct permutations. Let the given string have n_0 zeros and n_1 ones. We can sum the contributions by fixing the position p of the **first '1'**.

Consider any two positions j, k such that $p < j < k \leq n$. We count how many such permutations exist where $a_j = 0$ and $a_k = 1$.

- Positions $1 \ldots p-1$ are '0'.
- Position p is '1'.
- Position j is '0'.
- Position k is '1'.

We have fixed p zeros (at $1 \ldots p-1, j$) and 2 ones (at p, k). We have $n_0 - p$ zeros and $n_1 - 2$ ones remaining. These must be placed in the $n - p - 2$ remaining positions (all positions $> p$ except j and k). The number of ways to arrange these is $\binom{n-p-2}{n_1-2}$.

This is the number of permutations (with first '1' at p) that contribute 1 to the $g(a)$ sum for this *specific* pair (j, k). To get the total contribution for a fixed p, we sum this over all possible pairs (j, k) with $p < j < k$.

- The number of available positions after p is $n - p$.
- The number of ways to choose the pair (j, k) from these $n - p$ positions is $\binom{n-p}{2}$.
- So, the total contribution from all permutations with their first '1' at p is:

$$\text{Contrib}(p) = \binom{n-p}{2} \cdot \binom{n-p-2}{n_1-2}$$

The total answer is the sum of these contributions over all valid p:

$$\text{Ans} = \sum_{p=1}^{n_0} \left(\binom{n-p}{2} \cdot \binom{n-p-2}{n_1-2} \right)$$

Problem G. The Matrix

Problem Setter: Irfanur Rahman Rafio
Tester: Arnob Sarker, Muhiminul Islam Osim
Alt Writer: Rafid Bin Mostofa
Category: Bitmasks, Constructive

Hint 1: Try to understand the effect of one element in the total danger level of the Matrix.
Hint 2.1: XOR is an invertible operation.
Hint 2.2: Replacing a value is equivalent to applying a XOR update on it.
Hint 3: Precalculate the row and column danger levels.

Solution:

At first don't worry about which element to change. Just assume that you are changing the element at position (i, j) and see how it affects the total danger level of the Matrix.

The total danger level of the Matrix $= (r_1 + r_2 + \cdots + r_n) + (c_1 + c_2 + \cdots + c_m)$.
When you change an element $M_{i,j}$, the row danger level for all rows except the i-th row remain unchanged. Similarly, the column danger level for all columns except the j-th remain unchanged. Therefore, changing $M_{i,j}$ only affects r_i and c_j.

Next, try to quantify the effect of changing $M_{i,j}$ on r_i and c_j. Let, the new value of $M_{i,j}$ be $M'_{i,j}$. Since $a \oplus 0 = a$ and $a \oplus a = 0$, XOR is an invertible operation and the inverse operation of XOR is XOR itself. Now,

$$r_i = M_{i,1} \oplus M_{i,2} \oplus \cdots \oplus M_{i,m}$$

Cancelling out $M_{i,j}$ from both sides, you'll get:

$$r_i \oplus M_{i,j} = M_{i,1} \oplus M_{i,2} \oplus \cdots \oplus M_{i,j-1} \oplus M_{i,j+1} \oplus \cdots \oplus M_{i,m}$$

Now, adding $M'_{i,j}$ to both sides, you'll get:

$$r_i \oplus M_{i,j} \oplus M'_{i,j} = M_{i,1} \oplus M_{i,2} \oplus \cdots \oplus M_{i,j-1} \oplus M'_{i,j} \oplus M_{i,j+1} \oplus \cdots \oplus M_{i,m}$$

The right hand side is the new row danger level r'_i after changing $M_{i,j}$ to $M'_{i,j}$.

Therefore, you have:
The new row danger level for the i-th row, $r'_i = r_i \oplus M_{i,j} \oplus M'_{i,j}$.
Similarly, the new column danger level for the j-th column, $c'_j = c_j \oplus M_{i,j} \oplus M'_{i,j}$.

Let $x = M_{i,j} \oplus M'_{i,j}$.
Then, you can rewrite the above equations as:

$$r'_i = r_i \oplus x$$

$$c'_j = c_j \oplus x$$

So, the change in the total danger level of the Matrix due to changing $M_{i,j}$ to $M'_{i,j}$ is:

$$\Delta_{i,j} = (r'_i + c'_j) - (r_i + c_j) = (r_i \oplus x) + (c_j \oplus x) - (r_i + c_j)$$

This is now an optimization problem where you want to find the value of x that minimizes $\Delta_{i,j}$. For that, you need the x that minimizes $(r_i \oplus x) + (c_j \oplus x)$. Since the change in one bit doesn't affect the others, you can analyze it bit by bit. For each bit position k (from 0 to 29), you have the following cases:

- If the k-th bit of both r_i and c_j is 0, then the k-th bit of x should be 0 to minimize the contribution to the sum.
- If the k-th bit of both r_i and c_j is 1, then the k-th bit of x should be 1 to minimize the contribution to the sum.
- If the k-th bit of r_i is 1 and the k-th bit of c_j is 0, or vice versa, then the k-th bit of x does not matter, as it will contribute 2^k to the sum regardless.

This means that to minimize $(r_i \oplus x) + (c_j \oplus x)$, you should set the bits of x to match the bits of r_i and c_j where they are the same, and you can ignore the bits where they differ. You can easily see that a valid solution is to set $x = r_i \& c_j$. Another valid solution is $x = r_i | c_j$.

Now, where will you apply this change? You can simply try everywhere and choose the best position!
By precalculating all values of r_i and c_j, you can find $\Delta_{i,j}$ for a given position (i, j) in $\mathcal{O}(1)$. So, you can iterate through all elements of the Matrix, calculate the corresponding $\Delta_{i,j}$ for each element, and keep track of the minimum element in Δ. Finally, you can compute the minimum total danger level achievable by adding this minimum value (which is negative or 0) to the initial total danger level of the Matrix.

Simple Exercise: Prove that the minimum value of $\Delta_{i,j}$ is $(r_i \& c_j) \times (-2)$.

Problem H. Chemical Reaction

Problem Setter: Ashraful Islam Shovon
Tester: Rafid Bin Mostofa
Alt Writer: Irfanur Rahman Rafio
Category: Graphs

For each second, iterate over all the reaction rules and check if they produce a new chemical. The easy way to do that is to check if both x and y chemicals already exist in the chamber for a certain reaction rule (x, y, z) and if z does not exist yet. If both x and y exist and z does not, add z to the chamber.

For checking the existence of chemicals, using a set will introduce a logarithmic factor in the complexity. Instead, all the chemicals can be relabelled to the range $[0, n + 3 * m)$ and then a boolean array (or bitset) can be used to keep track of which chemicals are present in the chamber.
This way, checking the existence of chemicals x and y and adding z can be done in constant time. And for each second, all m reaction rules are checked. So the overall complexity for one second is $O(m)$.

It is easy to understand that if no new chemical is produced in a second, then no new chemical will be produced in any subsequent seconds. Thus, the process can be stopped when no new chemical is produced in a second. The implementation will look a bit like Bellman-Ford algorithm.

It can be shown that at most m seconds are needed. Thus, the overall complexity is $O(m^2 + n)$.

Notes:

- The reason why we need at most m seconds is that in each second, at least one new chemical is produced (otherwise we stop). Since each reaction rule can produce at most one new chemical, we can have at most m new chemicals.
- By using Kahn's algorithm for topological sorting, the complexity can be reduced to $O(n+m)$. However, since m is at most 1000, the above solution is sufficient.