ICPC 2017 Regional Contest
Jakarta Site

Problem Analysis

	PROBLEM TITLE	PROBLEM AUTHOR	ANALYSIS AUTHOR
A	Winning ICPC	Jonathan Irvin Gunawan	Jonathan Irvin Gunawan
B	Travelling Businessmen Problem	Alham Fikri Aji	Alham Fikri Aji
C	Non-Interactive Guessing Number	Ammar Fathin Sabili	Ammar Fathin Sabili
D	Make a Forest	Winardi Kurniawan	Suhendry Effendy
E	Parks of Jakarta	Ivan Adrian Koswara	Ivan Adrian Koswara
F	Random Number Generator	Ammar Fathin Sabili	Ammar Fathin Sabili
G	National Disaster: Two Towers	Vincentius Madya Putra Indramawan	Felix Halim
H	ANTS	Suhendry Effendy	Suhendry Effendy
I	XEN 3166	Ilham Winata Kurnia	Jonathan Irvin Gunawan
J	Meeting	Jonathan Irvin Gunawan	Jonathan Irvin Gunawan
K	Permutation	Ivan Adrian Koswara	Ivan Adrian Koswara
L	Sacred Scarecrows	Ashar Fuadi	Ashar Fuadi

A. Winning ICPC

This is the easiest problem in this contest.

For each team, we can count the number of problems solved by the team. We loop from the first team to the last team while keeping the index of the winning team so far and updating the winning team only if the current team solved more problems than the current winning team.

At the end of the loop, we can print the index of the winning team.

B. Travelling Businessmen Problem

First, we need to check whether the graph is a bipartite graph.

If the graph is not bipartite, there will be a path starts from a node A and ends in itself with odd distance. Therefore, for each case, we can always make the bussinessmen to meet at the same node. Hence, the answer is always 0.

Let assume the graph is bipartite. Assume we color the node with red and blue. If the initial starting points of the bussinessmen are from the same color, the answer will be 0 as we can move both bussinessmen to the same node. If the initial starting points are from different color, both bussinessman will never met. Hence, we need to find the minimum of abs(B[i] - B[j]) where i is a node with a red color and j is a node with a blue color.

To achieve this, we can put pair of node beauty and its color into a set. As the set is sorted, we can see that for any valid index i on set S, if S[i].color != S[i-1].color, we know that S[i].beauty - S[i-1].beauty is a candidate solution. For every candidate solution, we put them on a separate set, let say Z. Hence, for each question, we simply get the smallest value in set Z. For every update, we need to update set S and Z appropiately.

C. Non-Interactive Guessing Number

Let's start with the fact that the value of K in input is not important at all. Let L be the length of the string instead.

There are some trivial (or tricky?) cases:

- If the string ends with '=' and L > N, then there is no solution.
- If the string doesn't end with '=' and L ≥ N, then there is no solution.
- If the string doesn't end with '=' and L < N, then there is a simple solution: one by one from the first to last character, output the largest possible number if it is '<' or smallest possible number if it is '>'. For example, N = 10 and S = "><<><" will construct "1~10~9~2~8".

The main problem is how to construct a solution given that the string ends with '=' and L ≤ N, means that Romanos has to win on exactly the L-th turn given the scenario of the string. It is provable that if Romanos is actually able to win on X-th turn, then he can make himself win on Y-th turn where X ≤ Y

≤ L. Hence, we can simulate the best strategy to find out if Romanos can win on the L-th turn; which is the binary search strategy.

Let boolean function isPossible(A, B, pos) be the function returning if it is possible to construct a solution given that the smallest and largest possible numbers are A and B respectively and now we're on the pos-th turn (of L, 0-indexed). Based on previous proof, binary search can be used for this function. Let's call the number used to guess is mid; if the pos-th character is '<' then the number of numbers less than mid has to ≤ the number of numbers more than mid, else (if the character is '>') then the number of numbers less than mid has to > the number of numbers more than mid. It turns out the value of mid is floor((A + B + 1) / 2) if '<' or floor((A + B - 1) / 2) if '>'. The algorithm will be like this:

```
bool isPossible(long long A, long long B, int pos) {
    if (A == B) return true;
    if (S[pos] == '=') return false;

    if (S[pos] == '<') {
        long long mid = (A + B + 1) / 2;
        return isPossible(A, mid-1, pos+1);
    } else {
        long long mid = (A + B - 1) / 2;
        return isPossible(mid+1, B, pos+1);
    }
}
```

Calling isPossible(1, N, 0) will check if there exists a solution, but how can we construct the solution? There are actually two strategies that can be used by Romanos on each turn: using best strategy which is using binary search, or the "nonsense" one which is "the largest possible number if it is '<' or smallest possible number if it is '>'". Those two strategies are working dependently: in case Romanos needs to guess fast (L is small) he needs first strategy, or in case of need to guess slow but sure (L is large) he needs second strategy. To not making the game ends faster than L turns, Romanos always chooses the second strategy whenever possible. These strategies are sufficient to construct one of possible solution.

Let solve(A, B, pos) be the procedure to construct a solution given that the smallest and largest possible numbers are A and B respectively and now we're on the pos-th turn (of L, 0-indexed). Calling solve(1, N, 0) will do. The algorithm will be like this:

```
void solve(long long A, long long B, int pos) {
    if (S[pos] == '=') {
        guess(A);
    } else if (S[pos] == '<' && isPossible(A, B-1, pos+1)) {
        guess(B);
        solve(A, B-1, pos+1);
    } else if (S[pos] == '<') {
        long long mid = (A + B + 1) / 2;
        guess(mid);
        solve(A, mid-1, pos+1);
    } else if (S[pos] == '>' && isPossible(A+1, B, pos+1)) {
        guess(A);
        solve(A+1, B, pos+1);
    } else if (S[pos] == '>') {
        long long mid = (A + B - 1) / 2;
        guess(mid);
```

```
        solve(mid+1, B, pos+1);
    }
}
```

D. Make a Forest

The problem description makes the problem looks harder than it should be. Once you understand the problem better, you will realise that it actually is a fairly easy problem.

The first three requirements simply denote a forest with rooted trees where each vertex and edge has a label/value (not necessarily unique). Now, let's observe the following four points:

(a) There are N tuples (u_i, v_i, w_i),
(b) Each tuple appears exactly once as an edge in the forest,
(c) No two tuples have the same w_i,
(d) The forest should contain exactly N edges.

These points imply that there is a one-to-one relationship between each tuple and edge in the forest. So, processing a tuple is equal to adding an edge (and vertex) in the forest.

Requirement #5 forces us to prioritise tuples with smaller w_i to be closer to the root than tuples with larger w_i. We can construct the same forest by processing each tuple one by one in increasing order of w_i. When we process a tuple (u_i, v_i), add a new vertex with label v_i as a child of an existing vertex with label u_i which still has fewer than M children, or create a new vertex with label u_i as a root of a new tree and add vertex with label v_i as its child.

The key observation is: It doesn't matter which vertex with label u_i we choose when we process the tuple (u_i, v_i), as long as the chosen vertex still has fewer than M children; otherwise, a new tree is inevitable (greedy solution).

Solution: Sort all the tuples (u_i, v_i, w_i) based on w_i in ascending order. Add each tuple in this order into the forest one at a time, i.e. add a new vertex with label v_i as a child of *any* existing vertex with label u_i which still has fewer than M children; if there is no such vertex, then create a new tree with label u_i as its root, and put vertex with label v_i as its child.

Note that when implementing the solution, we don't need to actually build the forest. We only need to keep track the number of trees, the number of node for each label, and the number of existing child for each label. Alternatively, the last two can be combined into the number of possible new child for each label.

Here is one implementation in C/C++. The time-complexity for this solution is O(N lg N).

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

struct tuple { int u, v, w; };
bool operator < (const tuple &x, const tuple &y) { return x.w < y.w; };

int N, M;
tuple arr[100005];
map <int,int> cap;
```

```
int main() {
  scanf( "%d %d", &N, &M );
  for ( int i = 0; i < N; i++ )
    scanf( "%d %d %d", &arr[i].u, &arr[i].v, &arr[i].w );
  sort(arr,arr+N);
  int ans = 0;
  for ( int i = 0; i < N; i++ ) {
    if ( cap[arr[i].u] == 0 ) {
      cap[arr[i].u] += M - 1;
      cap[arr[i].v] += M;
      ans++;
    }
    else {
      cap[arr[i].u]--;
      cap[arr[i].v] += M;
    }
  }
  printf( "%d\n", ans );
  return 0;
}
```

E. Parks of Jakarta

Notice that this is essentially Tower of Hanoi, but with a couple of twists. First, the positions of the disks (the stones) aren't necessarily in a single pile (park); they may be distributed anywhere. Second, the cost of moving a stone is not necessarily the same between piles, and so the standard solution of solving Tower of Hanoi problems must be adapted. Finally, there are many configurations to visit, with an option of what the final configuration can be; this adds an extra layer on top of the problem.

First, observe that this problem is made of two parts. The first, small-scale part is how to efficiently bring configuration A to configuration B, for any two configurations of stones. The second, large-scale part is how to solve the actual problem: how to visit all intermediate configurations and then gather them all in one park.

We will figure out how to solve the small-scale part. The idea, as in Tower of Hanoi problems in general, is to notice the following fact: in order to move stone number K from a park to another, we need all stones 1, 2, ..., K-1 to be gathered in the third park. This gives a nice dynamic programming solution: for any K, most of the time we want to gather stones 1, 2, ..., K into a single park, and in doing so we don't need to care about the larger stones.

We use dynamic programming to solve the following problem: "What's the optimal cost to bring disks 1, 2, ..., K from a configuration P to a configuration Q?" Obviously, if P and Q can be any configuration, there are a lot of states to keep track of. The trick is to notice that there are only four possible P's that matter, and also only four possible Q's. The configuration "all in Park 1", "all in Park 2", and "all in Park 3" are some of the options for P and Q. In addition, the starting configuration, cut to include only stones up to K, is another option for P; likewise, the ending configuration is another option for Q.

However, these are all. It's never beneficial to leave stones in any other configuration; if we leave stones 1, 2, ..., K in configuration R that is not one of these, we won't be able to move any stone greater than K, and so we must continue moving configuration R to another configuration without being able to do any meaningful work in between. On the other hand, leaving all stones in one pile allows us to move stone K+1 from one of the other two piles to the other, and starting with all stones in configuration A and ending with all stones in configuration B are exactly what we're looking for, so these states are important.

The DP states will be recorded as dp[K][P][Q], where K indicates how many stones are being considered, and P and Q are state numbers: 1, 2, 3 for all in park of the same number, 0 for starting configuration (only for P), and 4 for ending configuration (only for Q). The base case is dp[0][P][Q] = 0; we never need to get any cost for moving zero stones. Now we set up a recurrence.

For ease, we will first assume P and Q are some of 1, 2, and 3; that is, all stones are in one park and we want to move all of them to another. We will adapt it to the cases 0 and 4 later. If P = Q, we're already done (and the cost is 0), so assume they are different and let R be the remaining park.

In order to move stones 1, 2, ..., K from P to Q, we need to move stone K from P to Q. There are two ways to do this: either directly, or going through R. (It might be cheaper to go through R! This is different from standard Tower of Hanoi.) In either case, to move stone K, we need all stones 1, 2, ..., K-1 to go to the third park.

The first method will incur the following cost:

- Move stones 1, 2, ..., K-1 from P to R: dp[K-1][P][R]
- Move stone K from P to Q: cost[P][Q] ($R_{P,Q}$ in description)
- Move stones 1, 2, ..., K-1 from R to Q: dp[K-1][R][Q]

They add up to dp[K-1][P][R] + cost[P][Q] + dp[K-1][R][Q]. Similarly, the second method costs dp[K-1][P][Q] + cost[P][R] + dp[K-1][Q][P] + cost[R][Q] + dp[K-1][P][Q]. We simply test both of these and take the minimum; that's the value of dp[K-1][P][Q].

What if P = 0? Stone K will still start from a particular park, say P_0, and from that we can still find R. Then the cost of the second method changes subtly, since we want to move to park P_0: it becomes dp[K-1][0][Q] + cost[P_0][R] + dp[K-1][Q][P_0] + cost[R][Q] + dp[K-1][P_0][Q]. The other cases are similar; at worst we'll just have several implications of very similar lines.

Finally, we just run this in order for K = 0, 1, 2, ..., N, and our result is stored in dp[N][0][4]. Since the numbers of options for P and Q are constant, and each evaluation is constant, this takes O(N) time. With N = 40, this is a breeze and we can repeat this quite a lot of times.

Which is good, since we'll need it for the second part. The idea is to first consider the following M+4 configurations: the initial state, the M intermediate states, and the 3 possible ending states: all in Park 1, all in Park 2, and all in Park 3. We consider all pairwise optimal costs between these, from any single configuration to any of the others. With M = 16, we can definitely take the $O(NM^2)$ time needed to find all of these costs. This is a preprocessing cost; we don't ever need to recompute these values any more.

Once we have these costs, we're now basically solving a variant of the Traveling Salesman Problem: starting from the starting configuration, we want to visit all intermediate configurations in some order, and reach one of the three ending configurations, with as low cost as possible. This can be solved using standard methods. One example is to use bitmask-based dynamic programming, using two

states: "current configuration" and "which intermediate configurations have been visited". This will have O(M 2^M) states, but with M = 16 this is not a problem.

F. Random Number Generator

Let's start with a simple Dynamic Programming (DP) solution. We didn't actually care about which numbers has already been called once, twice (or more), or not at all; what we really need are only the number (a.k.a. how many numbers) of them. Denote $DP_1(cd_0, cd_1, cd_2)$ be the expected number of calling until all numbers are called minimum twice, given that there are cd_0 numbers that has not been called at all, cd_1 numbers that has been called once, and cd_2 numbers that has been called twice or more. The formulation will be like this:

$$DP_1(0,0,N) = 0$$
$$DP_1(cd_0, cd_1, cd_2) = 1 + \frac{cd_0}{N} \times DP_1(cd_0 - 1, cd_1 + 1, cd_2) + \frac{cd_1}{N} \times DP_1(cd_0, cd_1 - 1, cd_2 + 1) + \frac{cd_2}{N} \times DP_1(cd_0, cd_1, cd_2)$$

Take a note that there is a cyclic recursion on second equation: $DP_1(cd_0, cd_1, cd_2)$ calls $DP_1(cd_0, cd_1, cd_2)$. We can rewrite the formula by moving them to Left Hand Side.

$$DP_1(cd_0, cd_1, cd_2) = 1 + \frac{cd_0}{N} \times DP_1(cd_0 - 1, cd_1 + 1, cd_2) + \frac{cd_1}{N} \times DP_1(cd_0, cd_1 - 1, cd_2 + 1) + \frac{cd_2}{N} \times DP_1(cd_0, cd_1, cd_2)$$
$$(1 - \frac{cd_2}{N}) \times DP_1(cd_0, cd_1, cd_2) = 1 + \frac{cd_0}{N} \times DP_1(cd_0 - 1, cd_1 + 1, cd_2) + \frac{cd_1}{N} \times DP_1(cd_0, cd_1 - 1, cd_2 + 1)$$
$$DP_1(cd_0, cd_1, cd_2) = (\frac{N}{N - cd_2}) \times (1 + \frac{cd_0}{N} \times DP_1(cd_0 - 1, cd_1 + 1, cd_2) + \frac{cd_1}{N} \times DP_1(cd_0, cd_1 - 1, cd_2 + 1))$$
$$DP_1(cd_0, cd_1, cd_2) = \frac{N}{N - cd_2} + \frac{cd_0}{N - cd_2} \times DP_1(cd_0 - 1, cd_1 + 1, cd_2) + \frac{cd_1}{N - cd_2} \times DP_1(cd_0, cd_1 - 1, cd_2 + 1)$$

The DP formulation is complete now, but it takes up to $O(N^3)$ complexity on memory. A trick to make it more efficient is to reduce the state number. Note that $cd_0 + cd_1 + cd_2$ is always N, then we can remove one of the state, i.e. cd_2. Denote $DP_2(cd_0, cd_1)$ be the expected number of calling until all numbers are called minimum twice, given that there are cd_0 numbers that has not been called at all and cd_1 numbers that has been called once. The formulation will be like this:

$$DP_2(0,0) = 0$$
$$DP_2(cd_0, cd_1) = \frac{N}{cd_0 + cd_1} + \frac{cd_0}{cd_0 + cd_1} \times DP_2(cd_0 - 1, cd_1 + 1) + \frac{cd_1}{cd_0 + cd_1} \times DP_2(cd_0, cd_1 - 1)$$

Above DP will run in $O(N^2)$ complexity. Unfortunately, running it on each testcase will takes $O(TN^2)$; even reusing the DP on each distinct value of N of the testcases will takes $O(N^3)$.

As long as there is variable N inside the DP, we can't reuse the DP for all testcases. There is another trick that can be done: divide the DP equations by N. It turns out that the N will disappear completely from the DP. Let's denote the formulation after dividing by N by DP_3 with these following formulation:

$$DP_3(0,0) = \frac{DP_2(0,0)}{N}$$
$$DP_3(0,0) = 0$$
$$DP_3(cd_0, cd_1) = \frac{DP_2(cd_0, cd_1)}{N}$$
$$DP_3(cd_0, cd_1) = \frac{1}{N} \times (\frac{N}{cd_0 + cd_1} + \frac{cd_0}{cd_0 + cd_1} \times DP_2(cd_0 - 1, cd_1 + 1) + \frac{cd_1}{cd_0 + cd_1} \times DP_2(cd_0, cd_1 - 1))$$
$$DP_3(cd_0, cd_1) = \frac{1}{cd_0 + cd_1} + \frac{cd_0}{cd_0 + cd_1} \times \frac{DP_2(cd_0 - 1, cd_1 + 1)}{N} + \frac{cd_1}{cd_0 + cd_1} \times \frac{DP_2(cd_0, cd_1 - 1)}{N}$$
$$DP_3(cd_0, cd_1) = \frac{1}{cd_0 + cd_1} + \frac{cd_0}{cd_0 + cd_1} \times DP_3(cd_0 - 1, cd_1 + 1) + \frac{cd_1}{cd_0 + cd_1} \times DP_3(cd_0, cd_1 - 1)$$

Finally, the rests are simple. DP_3 can be reused without resetting it for each testcases. For the i-th testcase, count how many numbers that has not been called at all, as X_i, and how many numbers that has been called once, as Y_i. The answer will be $N_i \times DP_3(X_i, Y_i)$ where N_i is the value of N on i-th testcase.

G. National Disaster: Two Towers

Determining whether there exists a **safe** path (that does not cross with any circle) between the two towers is hard (note that touching the circle is not considered as crossing it). It is easier to determine the opposite:

A **safe path does not exists** when there is a set of **connected** circles that connects the left or top side of the rectangle to the right or bottom side of the rectangle.

However there is a catch, a circle should only be considered **connected** to another circle if:
- One of the intersection points is **strictly** inside the rectangle, or
- Both intersection points are inside (not necessarily strictly inside) the rectangle.

Otherwise, it fails on corner cases like these:

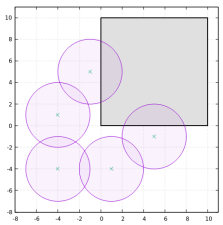
Fig 1: YES

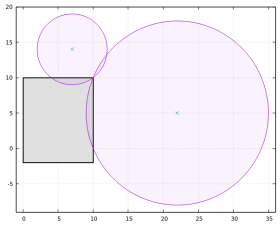
Fig 2: YES

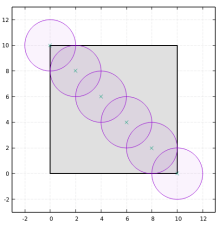
Fig 3: NO

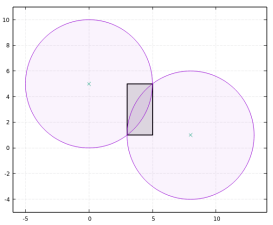
Fig 4: NO

Fig 1: shows the catch where naively connecting the circles that go outside the rectangle fails.
Fig 2: one of the circle intersection points is outside the rectangle and the other is touching the tower (not strictly inside the rectangle), thus the two circles are not considered connected.
Fig 3: the intersection points of the top left circle with the next circle are both inside the rectangle boundary (not necessarily strictly inside) and thus they are considered connected.

Fig 4: both intersections are inside (i.e., they are touching) the rectangle, thus considered connected.

The implementation to check whether two circles are connected requires computational geometry knowledge to find the actual intersections points between two circles.

One way to determine whether a safe path does not exists is to use disjoint-set (or union-find) algorithm:

1. Initially, there are N + 4 individual components: the N circles and 4 line segments of the rectangles
2. Merge connected-circles into the same component
3. Merge each of the four line segments with the component of a circle if any part of the **line segment** is **strictly** inside the circle
4. Output "NO" if the left or top line segment of the rectangle is in the same component with the right or bottom of the line segment of the rectangle. Otherwise, output "YES".

Be careful when implementing step 3 of the algorithm (determining whether a line segment intersect with a circle), otherwise, you may fail on cases such as in Fig 5 and Fig 6.

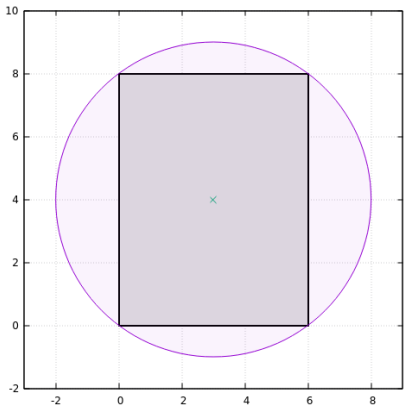

Fig 5: NO

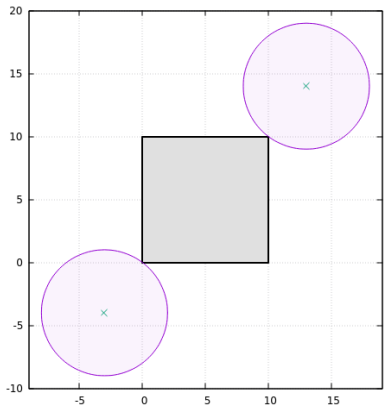

Fig 6: YES

Another corner case is when a circle is fully inside another circle as shown in Fig 7. For such case, ignore the inner circle. Otherwise, the inner circles may unwittingly connects circles outside the rectangle.

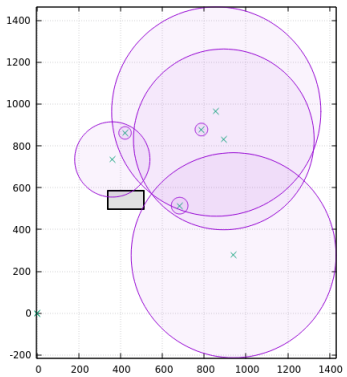

Fig 7: YES

H. ANTS

The query in this problem is asking for the cost of the optimal meeting point for any given K vertices on a tree (uniform cost). Given a set of K vertices on a tree, the cost of a meeting point (vertex) is the total distance from each of the K vertices to the meeting point.

In this editorial, we omit the proof for the solution and left it as an exercise to any interested reader.

The notion of optimal meeting point does not depend on whether the tree is rooted or unrooted, only the total distance matters. So, let's make the tree rooted as there are some calculations which can be done in a rooted tree, e.g., lowest common ancestor (LCA), which will be used in the solution. Simply choose any arbitrary vertex as the root.

K = 1 is a special case and trivial, thus, we remove the analysis for K = 1. The remaining of this editorial assumes K >= 2.

Let S be the set of K vertices in the query. Let T be the set of LCA vertices from any two vertices in S; there are $O(K^2)$ pairs of vertices in S.

Lemma-1: T contains an optimal meeting point for S.

Lemma-1 implies that we only need to check all the vertices in T and find the one with the smallest cost. Checking the distance between two vertices in a tree can be done with LCA query, i.e. in O(lg N), or O(1) with a sparse table. Hence, evaluating a meeting point (a vertex in T) can be done in O(K), i.e. find the distances from the meeting point to all vertices in S.

Now, here is the cool part.

Lemma-2: $|T| < |S|$.

You can use mathematical induction to prove Lemma-2: Adding one vertex into S will only add at most one extra vertex into T. Also, observe that $|T| = 1$ when $|S| = 2$.

With Lemma-2, finding the optimal meeting point can be done in $O(K^2)$, instead of $O(K^3)$.

I. XEN 3166

For easier discussion, let us assume the countries are sorted in the ascending order of country name.

For each country, we want to greedily assign the lexicographically smallest possible country code, but still lexicographically larger than the previous country's country code, so that the next countries can have more valid country code possibilities. Therefore, we can loop the countries while keeping track of the last country's country code, and for the current country, we generate the lexicographically smallest possible country code that is still lexicographically larger than the last country's country code.

How to achieve that? Let S be the current country name, T' be the last country code, and T is the current country code that we want to generate. Since T > T', then there exists an integer k where $T[k] > T'[k]$ and $T[i] = T'[i]$ for all $1 \le i \le k$. Since we want to make T as lexicographically small as possible,

we want to maximise the value of k. In other words, we want the common prefix of T and T' to be as long as possible.

We can check whether it is possible for T and T' to have a common prefix of length l by checking whether we can construct a subsequence of S which starts with T[1..l] + c for some c > T[l + 1]. To do this, we can define a function nxt(c, i) be the index of the next occurrence of character c in S after the i-th index, and then checking whether nxt(c, nxt(T'[l], nxt(T'[l - 1], ..., nxt(T'[1], 0) ...))) is less than some upper limit.

Once we found the maximum l possible, and the corresponding smallest c that makes l possible, we can greedily construct the lexicographically smallest subsequence that starts with T[1..l] + c to become T'.

J. Meeting

Note that this problem can be simplified as follows:

Given an array A where each element is between 0 to T. In one operation, we can increment or decrement the element in the array, but must still be between 0 to T. Determine the minimum number of operations such that the K-th smallest element is different from the (K+1)-th smallest element in the array.

Let f(A) be the largest possible integer and g(A) be the smallest possible integer such that:

- f(A) ≤ K and the f(A)-th smallest element is different from the (f(A)+1)-th smallest element in the array.
- g(A) ≥ K and the g(A)-th smallest element is different from the (g(A)+1)-th smallest element in the array.

If f(A) = g(A) = K, then we are happy. Otherwise, we want to change A into another array A' with the minimum cost such that f(A') = g(A') = K. There are two possible ways to achieve this:

- Decrement each element from the (f(A)+1)-th smallest element to the K-th smallest element by one, if the element is still greater than 0.
- Increment each element from the K-th smallest element to the g(A)-th smallest element by one, if the element is still less than T.

Among the possible ways, we use the one with less number of operations. If neither of the ways is possible, then we print -1.

K. Permutation

For convenience, we will define I to be the identity permutation, redefine M to be the smallest positive integer such that $P^M = I$ (instead of $P^M = P$), and redefine K to be one less than the input (in other words, we number the permutations in the list as 0, 1, 2, ..., M-1 instead of 1, 2, 3, ..., M).

One very important observation about permutations is that we can decompose it into disjoint cycles. For example, the permutation [2, 4, 5, 1, 3] is made of two cycles: (1, 2, 4) and (3, 5). (The cycle (1, 2,

4) means 1 maps to 2, 2 maps to 4, and 4 maps back to the beginning of the cycle 1.) Then, when we raise a permutation to the T-th power, we're simply raising each cycle into the T-th power, and it's easy to raise a cycle to the T-th power; given a cycle $(A_0, A_1, ..., A_{L-1})$, if we raise it to the T-th power, element A_I will now be mapped to $A_{(I+T) \bmod L}$.

The first question is, what is the value of M? Note that if we raise a cycle of length L to the L- th power, it will become the identity; all elements are mapped to themselves. And moreover, only multiples of L will bring this cycle to the identity. Thus, for every cycle in P of length L, we need L to divide M, otherwise this cycle won't become the identity. Thus we need M to be a multiple of the LCM of all cycle lengths. Conversely, taking M to be this LCM is enough.

Can we simply make the list of all permutations and sort it? As it turns out, no, since M can be as large as 232,792,560. So we need to obtain the value of T for each K directly without sorting the list of permutations.

Since we're sorting in lexicographic order, the initial elements matter the most. The list will begin with those where the first element is 1, followed by those where the first element is 2, then those where the first element is 3, and so on. The important thing is to be able to compute, exactly how many permutations begin with a specific first element.

Consider the cycle containing the first element. Only these elements will ever possibly appear as the first element. For example, if the cycle is (1, 2, 4) like in the example above, then only 1, 2, 4 can possibly be the first element; it will never be 3 or 5. Moreover, for each element that can possibly appear, they will appear the same number of times; this is easy to see by checking that P^1, P^2, P^3, ..., P^M cycles the first element. This means we can easily fix the first element. Suppose the cycle containing the first element has length L, and let X = M/L. Then take K = YX + Z where $0 \le Z < X$. This means we skip the first Y candidates (from smallest to largest), then take the next candidate as the first element. Moreover, from the rest of the elements, we're trying to find the Z-th permutation made from them.

For example, consider our example [2, 4, 5, 1, 3]; we have M = 6 and L = 3, so X = 2. Let K = 4 (the input is 5); then Y = 2 and Z = 0. So we skip the first Y = 2 possibilities of the first element, and take the next one. That is, we skip 1 and 2, and take 4 as the first element of the permutation. And moreover, we're looking for the 0-th (remember, zero-indexed) permutation that begins with the element 4.

We can in fact continue this way, but we need to be careful. After we fix the first element, the entire cycle containing it is also fixed. Moreover, for the later elements, there might be elements that end up impossible. For example, consider the permutation [2, 1, 4, 5, 6, 3]. Here we have M = 4. If the first element turns out to be 2, then the third element can only be 4 or 6; it's impossible to get 3 or 5. Conversely, if the first element is 1, then the third element can only be 3 or 5, not 4 or 6.

By carefully walking through the elements of the permutation, it's possible to obtain a solution in $O(N^2)$ per query. But this is not fast enough; we will need to do some preprocessing so we can improve this solution.

The idea is to store which possibilities are possible for each element, and how it affects the future elements. For example, in the example [2, 1, 4, 5, 6, 3], the first element has possibilities {1, 2}. This will completely fix the second element; moreover, if 1 is chosen, then the third element has possibilities {3, 5}, while if 2 is chosen, then the third element has possibilities {4, 6}. Not only that, we add some other information: if 1 was chosen, then T = 0 mod 2 (we need an even T in order for P^T to

have 1 at first element), and if 2 was chosen, then T = 1 mod 2. And similarly, if 3, 4, 5, 6 are chosen as the third element, then T = 0, 1, 2, 3 mod 4 respectively.

If we have all this information, we can solve a query in O(N) time. Given an input K, we first decompose it as YX + Z as above. The value of Y will decide which element is the first element, and what congruence relation that T must satisfy; the value of Z will be passed down the permutation. Using this first element, we can decide which list of possibilities we look at using Z, we can decide which element actually goes in that position.

In our example above, the third element has possibilities {3, 5} (if the first element is 1) or {4, 6} (if the first element is 2). Suppose the first element is 1, and Z = 1. Then we look at the list {3, 5}, and take the 1-th (zero-indexed) smallest element; that is, 5 goes to the third element. If instead the first element was 2 but with same Z, we would have taken 6 instead.

We will be able to reconstruct the permutation in O(N) time, and along the way we obtain several congruence relations for T. There are at most N of them (in fact, more careful analysis says that there are at most O(√N) of them, but we don't need it). Using Chinese Remainder Theorem, we can combine a pair of congruence relations into one in time O(log N) assuming there is no overflow, so in total we can combine all congruence relations into one in time O(N log N). Thus the total time per query would be O(N log N), which passes under the limit.

The problem is, how do we preprocess the permutation? The idea is to compute how much of the permutation has been fixed so far. Before we fix the first element, T can still be anything. After we fix the first element, whose cycle has length L, we would know the value of T mod L in the queries. Suppose we have a latter cycle of length L', and gcd(L, L') = G. Then we would also know the value of T mod G, since we know the value of T mod L. This means the first element of this latter cycle is somewhat fixed, since we already know the value of T mod G; there are less degrees of freedom.

This is exhibited in the above example, [2, 1, 4, 5, 6, 3]. The first cycle (1, 2) has length 2; this is L. The second cycle (3, 4, 5, 6) has length 4; this is L'. Whatever the value of T will end up to be, by inspecting K we will know what the first element is, and hence the value of T mod L = T mod 2. But this means the second cycle is not fully free; we know the value of T mod 2, and so not all possibilities for the second cycle remain. For example, if we have T mod 2 = 0, it's impossible to have T mod 4 = 1; the value of T mod 4 will only be either 0 or 2. As we see before, a starting element of 1 means the second cycle can only start with 3 or 5, not 4 or 6; this is where this observation occurs.

Thus, the idea is to keep track this modulo over the permutation. After preprocessing the i-th element, the value of T is fixed modulo B for some B. After looking at the next (i+1)-th element, it belongs in a cycle of length L; assume this is not a cycle that includes an earlier element (if it was, then we would have fixed the cycle). Define G = gcd(B, L); then we can partition the possibilities for the (i+1)-th element into G groups; one for T = 0 mod G, one for T = 1 mod G, and so on. Each of these groups forms a list. The example above had the groups {3, 5} and {4, 6} for the third element, obtained when T = 0 mod 2 and T = 1 mod 2 respectively. Partitioning the list is easy; simply check which element will occupy this (i+1)-th position when we raise P to the 1-st, 2-nd, 3-rd, ..., L-th power; if an element X occupies it when P is raised to the Y-th power, then X goes into the list T = Y mod G. Finally, after we partition all the elements, the new value of B is lcm(B, L); the value of T is now fixed that much.

This preprocessing in fact takes the same time O(N log N) as the time per query; most of the time is taken by computing the value of G, while partitioning the elements into lists takes only linear time. And given the above information, we can solve each query in O(N log N) time as described, which passes under the time limit.

L. Sacred Scarecrows

A sacred configuration satisfies the following two conditions:

1. Each row contains at least one scarecrow.
2. Each consecutive two columns contain at least one scarecrow.

Let $f(r)$ be the number of sacred configuration, considering only rows in the subset $r \in \{1,2,\cdots,R\}$ (i.e., we want to leave all rows not in r untouched).

This can be computed using a simple DP, where the state is <current column, does the previous column contain scarecrow>. For the transition, we can:
leave the current column untouched (only possible if the previous column contains scarecrows), or
put one or more scarecrows in the current column. If the current column contains k empty rows (considering only rows in r), then there are $2^k - 1$ ways.

Using the principle of inclusion-exclusion, the final answer is $\sum (-1)^{R-|r|} f(r)$

There are 2^R possible subsets of rows. The DP for computing f() for each subset takes O(C) time. When implemented efficiently, the final time complexity is $O(2^R \cdot C)$.