Contest 5A Analysis

14.03.2019

Gennady Korotkevich, Filipp Rukhovich

Moscow PreFinals Workshop 2019

A. Automaton

In this problem, we are to build an acyclic DFA for some string S which accepts all suffixes of given string S. Number of states should be as small as possible.

A. Automaton

Obviously, there should be at least $|S| + 1$ states in the resulting automaton which form a path of accepting the whole string S. We will build the automaton which uses only these $|S| + 1$ strings.

The algorithm can be built in the following way. Let $S = S[1..n]$. We will build a sequence of automatons for empty string, $S[1..1]$, $S[1..2]$, $S[1..3]$, ..., $S[1..n]$ such that sets of states and transitions of each previous automaton will be subsets of sets of states and transitions of next automaton.

Let $f(T)$ be a state which will be achieved if we go from start state with string T. Since T is accepted by the automaton, the state $f(T)$ will not be changed. Let also D be a (current) set of states, and for $d \in D$, $class(d)$ will be set of such substrings s of S that $f(s) = d$.

A. Automaton

Let also *start* be start state of automaton.

We build that automaton maintaining the following invariants after i-th step, $i = 0, 1, \ldots, |S|$:

- automaton accepts all substrings of string $S[1..i]$;
- let $L(d)$, $d \in D$, is a longest string in $class(d)$. Then, for any $d \in D \setminus \{start\}$, it's true that $class(d) = \{L(d), L(d)[2..|L(d)|], L(d)[3..|L(d)|], \ldots, L(d)[l..|L(d)|]\}$ for some $l = l(d)$, $l > 0$; in other words, $class(d)$ consist of such a string $L(d)$ and all its suffices which start not later than from l-th symbol;
- for each state $d \in D \setminus \{start\}$ we store link to state $suff = suff(d)$, such that $suff(d) = f(L(d)[l+1..|l(d)|])$.

A. Automaton

It gives the following way to move from $S[1..i-1]$ to $S[1..i]$, $i = 1, 2, \ldots, n$:

1. build new state $d(S[1..i])$ and edge from $d(S[1..i-1])$ to $d(S[1..i])$ with symbol $S[i]$;
2. $st = suff(d(S[1..i-1]))$;
3. for(; $st! = start$; $st = suff(st)$):
   - if we can go from st by $S[i]$ then break;
   - else build an edge from st to $d(S[1..i])$ with symbol $S[i]$.
4. if we can go from st by $S[i]$ to state d' then store d' as $suff(d(S[1..i]))$;
5. otherwise, store $start$ as $suff(d(S[1..i]))$ and build an edge from st to $d(S[1..i])$ with symbol $S[i]$.

A. Automaton

This algorithm maintains the invariant. It can be proved from the fact that $[d(S[1..i])]$ consists of only such a suffices of $S[1..i]$ which are not substrings of $S[1..i-1]$, and it satisfies second condition. It is very similar to classical algorithm of building a suffix automaton, but without cloning a vertex.

It can be proved as in this algorithm using amortized analysis (as in prefix function) that all iterations of *for* will add no more than $|S|$ edges in common; so, the automaton we built is what we need. The complexity of the solution is $O(|S|)$.

B. Bank of a River

George and Mary are going in a car along straight line. There are n turns; if they turn on i-th of them then the result of a game will be h_i; if they will not turn then the result will be h_{n+1}. The turns seem absolutely the same, and Mary can make George not to notice some of turns except last. So, George can only decide it is needed to turn on k-th, $1, 2, \ldots, n$, turn he noticed or not. You are to find a price of a game.

B. Bank of a River

Mixed strategy for George can be reformulate as if for each turn, he decided only whether he should turn or not; he turns in first turn he noticed with probability p_1, on second - with probability p_2 etc.

Mary can decide which turns George should see. Let G_k, $k = n+1, n, \ldots, 1$ be a game in which George has some classical strategy, but Mary knows that now they missed $i-1$ turns. Let V_k, $k = n+1, n, \ldots, 1$ be a price of G_k.

Obviously, $V_{n+1} = h_{n+1}$ (there are only one variant of result). Suppose that we know V_i for all $i > k$ and want to find V_k. Then, Mary should decide which k'-th turn George should see first. Suppose that p is a probability for George to turn on first after current moment turn. Then, the price will be $p * h_{k'} + (1-p) * V_{k'+1}$.

B. Bank of a River

For each p, Mary can take $f_k(p) = \min_{k \leqslant k' \leqslant n} \{p * h_{k'} + (1-p) * V_{k'+1}\}$. So, if George knows that he missed at least k turns, then he should take such a $p \in [0,1]$ that the $f_k(p)$ should be maximized.

In fact, $f_k(p)$ is convex-up and piecewise-linear. Let's store the segments in std::set (or it's analogues in other languages) being sorted by p, and their ends - in the second set sorted by increasing of value of function. Then on each step, maximum of function can be found in $O(log\ n)$; after that, we can add new line and erase one by one all segments and parts of segments which are higher than it is starting from the point having a tangent parallel to new line. The total complexity of the solution if $O(n\ log\ n)$.

C. Circles and Matrix

Given a rectangular $S \times S$, $S = 1000$, table and sequence of n circles $a_1, a_2, \ldots, a_n$ in it. Two circles a_i, a_j, $i < j$, are called incorrect if a_i and a_j have common points and at least one of circles a_k, $i < k \leqslant j$ isn't contained in a_i. The problem is to find any incorrect pair or print "Ok" if there are no such pairs.

Note that if we move horizontal scanning line from up to down then the intersection of the line with some circle of raduis r will be changed no more than $O(\sqrt{r})$ times. In give us a possibility to reveal whether two circles have common points or not or whether one of the circle is contained in another one in $O(\sqrt{S})$ time.

C. Circles and Matrix

The statement means that there are no incorrect pairs if and only if the following two conditions are satisfied:

1. for each $i \in \{1, 2, \ldots, n\}$, there exist such a number $R = R(i) \in \{i, i+1, \ldots, n\}$ such that a_i contains circles a_{i+1}, a_{i+2}, ..., $a_{R(i)}$ and doesn't have common points with circles $a_{R(i)+1}$, $a_{R(i)+2}$, ..., a_n;
2. for any $i, j \in \{1, 2, \ldots, n\}$, segments $[i, R(i)]$ and $[j, R(j)]$ don't have common points or one of them contains into another.

Suppose that these conditions are satisfied. Then, let $p(i)$, $i = 1, 2, \ldots, n$, is an nonnegative integer such that $p(i)$ is the maximum number of circle which is less than i and $a_{p(i)}$ contains a_i or 0 if there are no such an index. Numbers $p(i)$ forms a rooted forest on circles.

C. Circles and Matrix

$p(i)$ can be calculated in the following way. Let ST be a stack, initially empty, but containing some circles at each moment of time. Then, go over all the circles; for i-th of them, remove top circles of ST until ST becomes empty or current top circle contains a_i. Then, store index of top circle or 0 if ST is empty to $p(i)$ and then add a_i to ST. This part of algorithm works in $O(n\sqrt{S})$ time. Then, $R(i)$ will be last index of circle which lie into a subtree of i-th circle.

To finish the solution, it's enough to check that all roots are pairwise non-intersecting, and for each $i \in \{1, 2, \ldots, n\}$, children of i-th circle are pairwise non-intersecting to. Any subset of k circles, $k \in \{1, 2, \ldots, n\}$, can be checked to be pairwise non-intersecting using horizontal scanning line in $O(k\sqrt{k} log\ (k\sqrt{S})) = O(k\sqrt{S} log\ k) = O(k\sqrt{S} log\ n)$ time; so the total complexity of the solution will be $O(n\sqrt{S} log\ n)$.

C. Circles and Matrix

It can be sped up to $O(n\sqrt{S} + S + n\ log\ n)$ if one performs sortings of events in scanlines using one common bucket sort of all the events for all n circles.

D. Diagonals

Given a simple polygon $A = A_0 A_1 \dots A_{n-1}$ enumerated, without loss of generality, counterclockwise. You are to split it by diagonals into minimal possible number of convex polygon.

First of all, if A is convex then print 1. Otherwise, calculate for any i, j, $0 \leqslant i, j < n$, $i \neq j$ whether segment $A_i A_j$ is a diagonal or not. Then, for any **oriented** diagonal $A_i A_j$, let $d[i][j]$ be a minimal number of convex polygons we can split the polygon $A_i A_{i+1} \dots A_j$, if $i < j$, or $A_i A_{i+1} \dots A_{n-1} A_0 \dots A_j$, if $i > j$, into. Also, $d[i][i+1]$ should be equal to zero, so do $d[n-1][0]$.

At the same time, we will calculate $f[i][k][j]$, $0 \leqslant i, j, k < n$; here, $f[i][k][j]$ is equal to minus one plus the same as $d[i][j]$, but in assumption that $A_k A_j$ and $A_i A_j$ are sides of one of convex polygons. If $A_i A_j$ is not a diagonal then $f[i][k][j]$ for any k and $d[i][j]$ are both infinite.

D. Diagonals

We will calculate both $dp[i][j]$ and $f[i][k][j]$ in order of number of vertices in polygon corresponding with $dp[i][j]$; it's in fact length of "subsegment" of polygon from i-th vertex to j-th one.

Obviously, if $i < j$, then $dp[i][j] = 1$ if $j = i + 1$ and $\min_{k=i+1}^{j} 1 + f[i][k][j]$; by the same way, $dp[j][i]$ can be found from f-s. But how to calculate f?

To find $f[i][k][j]$, we should build a convex subsequence of points $i = i_0, i_1, i_2, \ldots, i_{z-1} = k, i_z = j$ on "subsegment" between i and j and summarize $dp[i_{l-1}][i_l]$ for $l = 1, 2, \ldots, z$. Then, $f[i][k][j] = d[k][j] + \min\{f[i][k'][k] \mid i < k' < k,\ A_{k'}A_k$ is a diagonal and angle $A_{k'}A_kA_j$ is oriented negatively$\}$. It gives us a solution with complexity $O(n^4)$.

D. Diagonals

To make it faster, for each point A_k sort all other points by polar angle. Then, after calculating $d[i][j]$, one can store all $f[i][k][j]$-s for all k in the array in such a way that for all future $f[i][j][j']$-s, the minimum described above could be found in $O(log\ n)$ or even $O(1)$ as minimum on prefix of array. It optimizes a solution to $O(n^3)$.

E. Empire and Roads

We are given a graph in a dynamic setting. The set of nodes V is fixed, but the edges of the graph are added and deleted over time.

We are to answer queries of the following type: given a set of nodes $\{u_1, u_2, \ldots, u_k\} \subseteq V$ and a moment of time, check whether these nodes form one or more whole connected components of the graph at this moment of time.

E. Empire and Roads

We present a randomized solution to this problem.

E. Empire and Roads

We present a randomized solution to this problem.

Let's pick a random integer weight for each edge.

E. Empire and Roads

We present a randomized solution to this problem.

Let's pick a random integer weight for each edge.

For each node $v \in V$, let $xor[v]$ be the xor-product of weights of all edges adjacent to this node. Note that these values can be updated in $\mathcal{O}(1)$ upon insertion and deletion of edges.

E. Empire and Roads

We present a randomized solution to this problem.

Let's pick a random integer weight for each edge.

For each node $v \in V$, let $xor[v]$ be the xor-product of weights of all edges adjacent to this node. Note that these values can be updated in $\mathcal{O}(1)$ upon insertion and deletion of edges.

Crucial observation: $\{u_1, u_2, \ldots, u_k\}$ forms a number of whole connected components if and only if each edge in the graph is adjacent to exactly zero or two of the nodes u_i.

E. Empire and Roads

To answer a query, we compute:

$$xor[u_1] \oplus xor[u_2] \oplus \ldots \oplus xor[u_k]$$

where $\oplus$ denotes the xor-operation.

E. Empire and Roads

To answer a query, we compute:

$$xor[u_1] \oplus xor[u_2] \oplus \ldots \oplus xor[u_k]$$

where $\oplus$ denotes the xor-operation.

If the above value is non-zero, we know that the answer is `NO`.

E. Empire and Roads

To answer a query, we compute:

$$xor[u_1] \oplus xor[u_2] \oplus \ldots \oplus xor[u_k]$$

where $\oplus$ denotes the xor-operation.

If the above value is non-zero, we know that the answer is `NO`.

Otherwise with high probability the answer is `YES`.

The probability of an incorrect positive answer is $\frac{1}{M}$, where M is the maximum weight of an edge. This probability is sufficiently small for $M = 2^{32}$ or $M = 2^{64}$.

E. Empire and Roads

To answer a query, we compute:

$$xor[u_1] \oplus xor[u_2] \oplus \ldots \oplus xor[u_k]$$

where $\oplus$ denotes the xor-operation.

If the above value is non-zero, we know that the answer is `NO`.

Otherwise with high probability the answer is `YES`.

The probability of an incorrect positive answer is $\frac{1}{M}$, where M is the maximum weight of an edge. This probability is sufficiently small for $M = 2^{32}$ or $M = 2^{64}$.

The whole solution works in linear time with respect to the size of the input.

F. Frogs

There is an infinitely long sequence of cells. For each $i \geqslant 0$, the beauty of cell i is equal to $x^i \bmod p$.

Initially k smart frogs (numbered 1 through k) are standing at cell 0, and each of them has happiness equal to 1.

They move repeatedly according to the following steps:

1. Frog 1 moves one cell forward, and its happiness increases by the beauty of the cell it enters.
2. For $i = 2, 3, \ldots, k$, if Frog $i - 1$ moves and the happiness of Frog $i - 1$ is a multiple of m, Frog i will move one cell forward and its happiness increases by the beauty of the cell it enters. Otherwise Frog i does nothing.
3. If the distance between Frog 1 and Frog k is more than or equal to d, the movement ends.

Compute the position of Frog 1 when they finish the movement.

F. Frogs

Let $f(t)$ be the position of Frog 2 at time t.

F. Frogs

Let $f(t)$ be the position of Frog 2 at time t.

Note that Frog 3 can move only if Frog 2 moves, and this process is similar to how Frog 2 moves with regard to Frog 1. It can be seen that the position of Frog 3 at time t is $f(f(t))$.

F. Frogs

Let $f(t)$ be the position of Frog 2 at time t.

Note that Frog 3 can move only if Frog 2 moves, and this process is similar to how Frog 2 moves with regard to Frog 1. It can be seen that the position of Frog 3 at time t is $f(f(t))$.

Similarly, the position of Frog 4 is $f(f(f(t)))$, ..., the position of Frog k is $f^{k-1}(t)$.

F. Frogs

Let $f(t)$ be the position of Frog 2 at time t.

Note that Frog 3 can move only if Frog 2 moves, and this process is similar to how Frog 2 moves with regard to Frog 1. It can be seen that the position of Frog 3 at time t is $f(f(t))$.

Similarly, the position of Frog 4 is $f(f(f(t)))$, ..., the position of Frog k is $f^{k-1}(t)$.

We are asked to compute the smallest t such that $t - f^{k-1}(t) \geqslant d$.

F. Frogs

Let $f(t)$ be the position of Frog 2 at time t.

Note that Frog 3 can move only if Frog 2 moves, and this process is similar to how Frog 2 moves with regard to Frog 1. It can be seen that the position of Frog 3 at time t is $f(f(t))$.

Similarly, the position of Frog 4 is $f(f(f(t)))$, ..., the position of Frog k is $f^{k-1}(t)$.

We are asked to compute the smallest t such that $t - f^{k-1}(t) \geqslant d$.

$t - f^{k-1}(t)$ is a non-decreasing function because for any t $f^{k-1}(t+1) - f^{k-1}(t)$ is 0 or 1.

Since $t - f^{k-1}(t)$ is monotonous, we can use binary search.

F. Frogs

The main part of the solution is computing $f(t)$.

F. Frogs

The main part of the solution is computing $f(t)$.

Let's call integer n *good* if the total beauty of cells $1, 2, \ldots, n$ is a multiple of m.

F. Frogs

The main part of the solution is computing $f(t)$.

Let's call integer n *good* if the total beauty of cells $1, 2, \ldots, n$ is a multiple of m.

Since x^i is periodic with period $p-1$, the total beauty of any $m(p-1)$ consecutive cells is divisible by m. Thus, n is good if and only if $n \bmod m(p-1)$ is good.

F. Frogs

The main part of the solution is computing $f(t)$.

Let's call integer n *good* if the total beauty of cells $1, 2, \ldots, n$ is a multiple of m.

Since x^i is periodic with period $p-1$, the total beauty of any $m(p-1)$ consecutive cells is divisible by m. Thus, n is good if and only if $n \bmod m(p-1)$ is good.

For each $n \leqslant m(p-1)$, precalculate whether n is good.

F. Frogs

The main part of the solution is computing $f(t)$.

Let's call integer n *good* if the total beauty of cells $1, 2, \ldots, n$ is a multiple of m.

Since x^i is periodic with period $p-1$, the total beauty of any $m(p-1)$ consecutive cells is divisible by m. Thus, n is good if and only if $n \bmod m(p-1)$ is good.

For each $n \leqslant m(p-1)$, precalculate whether n is good.

$f(t)$ is the number of good integers between 1 and t. We can calculate it in $\mathcal{O}(1)$ using prefix sums of the precalculated table.

G. Graph Modifications

We are given an undirected graph G satisfying the following properties:

1. G is simple, that is, it contains no self-loops or multiple edges.
2. G is connected.
3. G contains no simple cycles which have length at least 4.

Find the maximal number of edges one can add to G while keeping the properties above.

G. Graph Modifications

The properties in the statement are equivalent to the following:

G. Graph Modifications

The properties in the statement are equivalent to the following:

1. G has a spanning tree. Fix one spanning tree T, and let's call edges contained in T "tree edges", and call other edges "non-tree edges".

G. Graph Modifications

The properties in the statement are equivalent to the following:

1. G has a spanning tree. Fix one spanning tree T, and let's call edges contained in T "tree edges", and call other edges "non-tree edges".
2. If $e = (u, v)$ is a non-tree edge, the distance between u and v in T is exactly two.

G. Graph Modifications

The properties in the statement are equivalent to the following:

1. G has a spanning tree. Fix one spanning tree T, and let's call edges contained in T "tree edges", and call other edges "non-tree edges".
2. If $e = (u, v)$ is a non-tree edge, the distance between u and v in T is exactly two.
3. For each non-tree edge $e = (u, v)$, color the path between u and v in T. No edge is colored more than once.

G. Graph Modifications

The properties in the statement are equivalent to the following:

1. G has a spanning tree. Fix one spanning tree T, and let's call edges contained in T "tree edges", and call other edges "non-tree edges".
2. If $e = (u, v)$ is a non-tree edge, the distance between u and v in T is exactly two.
3. For each non-tree edge $e = (u, v)$, color the path between u and v in T. No edge is colored more than once.

The original problem can be reduced to the following problem:

G. Graph Modifications

The properties in the statement are equivalent to the following:

1. G has a spanning tree. Fix one spanning tree T, and let's call edges contained in T "tree edges", and call other edges "non-tree edges".
2. If $e = (u, v)$ is a non-tree edge, the distance between u and v in T is exactly two.
3. For each non-tree edge $e = (u, v)$, color the path between u and v in T. No edge is colored more than once.

The original problem can be reduced to the following problem:

You are given a tree, and some edges are already colored. In each operation, you must choose a path of length 2 and color it (you can't color an edge if the edge is already colored). How many operations can you perform?

G. Graph Modifications

The problem can be solved greedily.

G. Graph Modifications

The problem can be solved greedily.

Root the tree and find the deepest uncolored edge e.

G. Graph Modifications

The problem can be solved greedily.

Root the tree and find the deepest uncolored edge e.

If e is not adjacent to other uncolored edges, you can't color this edge, ignore it.

G. Graph Modifications

The problem can be solved greedily.

Root the tree and find the deepest uncolored edge e.

If e is not adjacent to other uncolored edges, you can't color this edge, ignore it.

If e is adjacent to one of its sibling edges (let's call it e_2), you should color e and e_2 at the same time (otherwise you can't color both).

G. Graph Modifications

The problem can be solved greedily.

Root the tree and find the deepest uncolored edge e.

If e is not adjacent to other uncolored edges, you can't color this edge, ignore it.

If e is adjacent to one of its sibling edges (let's call it e_2), you should color e and e_2 at the same time (otherwise you can't color both).

If e is adjacent only to its parent edge, you should color e and its parent at the same time.

G. Graph Modifications

Note that if there are no colored edges initially, we can always pair up all tree edges (except one if their number is odd) using our greedy algorithm.

G. Graph Modifications

Note that if there are no colored edges initially, we can always pair up all tree edges (except one if their number is odd) using our greedy algorithm.

It follows that the answer for a tree with n vertices and without colored edges is $\lfloor \frac{n-1}{2} \rfloor$.

G. Graph Modifications

Note that if there are no colored edges initially, we can always pair up all tree edges (except one if their number is odd) using our greedy algorithm.

It follows that the answer for a tree with n vertices and without colored edges is $\lfloor \frac{n-1}{2} \rfloor$.

Therefore, if we remove all colored edges and find the sizes of connected components $s_1, s_2, \ldots, s_k$, the answer is $\lfloor \frac{s_1-1}{2} \rfloor + \lfloor \frac{s_2-1}{2} \rfloor + \ldots + \lfloor \frac{s_k-1}{2} \rfloor$.

H. Hacked "Lines"

We have a 9×9 field of cells. Each cell is either empty or contains a ball of some color.

On each turn, we must put two new balls of any colors into two empty cells.

After each turn, if there is at least one horizontal, vertical, or diagonal sequence of 5 consecutive balls of the same color, all balls belonging to at least one such sequence are destroyed, and the game ends.

The number of destroyed balls has to be as large as possible.

H. Hacked “Lines”

The intended solution is brute-force search.

H. Hacked "Lines"

The intended solution is brute-force search.

Let's fix our last turn: two cells and the colors of balls put into these cells.

These two cells are the "centers" of balls destruction. We need to figure out how many cells in each of eight directions from each center will contain balls of the same color as the center.

It is important that no lines of 5 or more balls of the same color must appear before the last turn.

H. Hacked "Lines"

The intended solution is brute-force search.

Let's fix our last turn: two cells and the colors of balls put into these cells.

These two cells are the "centers" of balls destruction. We need to figure out how many cells in each of eight directions from each center will contain balls of the same color as the center.

It is important that no lines of 5 or more balls of the same color must appear before the last turn.

During the search, we can greedily estimate the maximum number of additional balls we can place that can be destroyed.

If the sum of the current number of balls to be destroyed in our construction and the greedy estimation does not exceed the current best result, we can exit this branch.

H. Hacked “Lines”

Another important optimization is *iterative deepening*.

Instead of looking for the best possible result, we will be looking for a result of at least X, increasing X step by step.

H. Hacked “Lines”

Another important optimization is *iterative deepening*.

Instead of looking for the best possible result, we will be looking for a result of at least X, increasing X step by step.

In this case, it can be inferred that one of two centers is responsible for destroying at least $\lceil \frac{X}{2} \rceil$ balls.

H. Hacked "Lines"

Another important optimization is *iterative deepening*.

Instead of looking for the best possible result, we will be looking for a result of at least X, increasing X step by step.

In this case, it can be inferred that one of two centers is responsible for destroying at least $\lceil \frac{X}{2} \rceil$ balls.

Thus, we can prune branches where the first center destroyed a small number of balls, but our greedy estimation tells that the second center can still destroy a lot.

H. Hacked “Lines”

Another important optimization is *iterative deepening*.

Instead of looking for the best possible result, we will be looking for a result of at least X, increasing X step by step.

In this case, it can be inferred that one of two centers is responsible for destroying at least $\lceil \frac{X}{2} \rceil$ balls.

Thus, we can prune branches where the first center destroyed a small number of balls, but our greedy estimation tells that the second center can still destroy a lot.

Also, if many balls were placed for destruction by the first center, our search will be more efficient for the second center, as there will be less empty space on the grid.

I. ICPC Abbreviations

After parsing the input, we are left with the following core problem:

Given n strings $s_1, \ldots s_n$ of total length N, find their nonempty prefixes $p_1, \ldots, p_n$ such that

$$p_1 p_2 \ldots p_n$$

is lexicographically minimal.

I. ICPC Abbreviations

Assume that we have already found $p_{k+1}, \ldots, p_n$ such that $p_{k+1} p_{k+2} \ldots p_n$ is lexicographically minimal.

I. ICPC Abbreviations

Assume that we have already found $p_{k+1}, \ldots, p_n$ such that $p_{k+1} p_{k+2} \ldots p_n$ is lexicographically minimal.

Note that such choice of $p_{k+1}, \ldots, p_n$ is optimal for any choice of $p_1, \ldots, p_k$.

This is true since $t < t'$ implies $wt < wt'$ for any string w.

I. ICPC Abbreviations

Assume that we have already found $p_{k+1}, \dots, p_n$ such that $p_{k+1}p_{k+2}\dots p_n$ is lexicographically minimal.

Note that such choice of $p_{k+1}, \dots, p_n$ is optimal for any choice of $p_1, \dots, p_k$.

This is true since $t < t'$ implies $wt < wt'$ for any string w.

Therefore, we can find optimal p_i for $i = n, n-1, \dots, 1$ one by one.

I. ICPC Abbreviations

How to efficiently find the optimal p_k, given $p_{k+1}, \ldots, p_n$?

I. ICPC Abbreviations

How to efficiently find the optimal p_k, given $p_{k+1}, \ldots, p_n$?

Let $t := p_{k+1}p_{k+2}\ldots p_n$. We want to find a nonempty prefix p of s_k such that pt is lexicographically minimal.

I. ICPC Abbreviations

How to efficiently find the optimal p_k, given $p_{k+1}, \ldots, p_n$?

Let $t := p_{k+1}p_{k+2}\ldots p_n$. We want to find a nonempty prefix p of s_k such that pt is lexicographically minimal.

We can efficiently compare pt and $p't$ using hashes:

- binary search for the longest common prefix of pt and $p't$, and
- compare the letters on the first position behind that prefix.

I. ICPC Abbreviations

How to efficiently find the optimal p_k, given $p_{k+1}, \dots, p_n$?

Let $t := p_{k+1} p_{k+2} \dots p_n$. We want to find a nonempty prefix p of s_k such that pt is lexicographically minimal.

We can efficiently compare pt and $p't$ using hashes:

- binary search for the longest common prefix of pt and $p't$, and
- compare the letters on the first position behind that prefix.

If we use randomized hashing, this algorithm works in $\mathcal{O}(\log N)$ time with very high probability. We need to perform $|s_k| - 1$ such comparisons to find p_k, yielding a total runtime of $\mathcal{O}(|s_k| \log N)$. Summing over all s_k, we obtain $\mathcal{O}(N \log N)$.

J. Jigsaw Puzzle

We have two $n \times m$ matrices filled with integers from 0 to $nm - 1$. A is filled in row-major order, and B is filled in column-major order:

$$A = \begin{matrix} 0 & 1 & \dots & m-1 \\ m & m+1 & \dots & 2m-1 \\ \dots & \dots & \dots & \dots \\ (n-1)m & (n-1)m+1 & \dots & nm-1 \end{matrix}$$

$$B = \begin{matrix} 0 & n & \dots & (m-1)n \\ 1 & n+1 & \dots & (m-1)n+1 \\ \dots & \dots & \dots & \dots \\ n-1 & 2n-1 & \dots & mn-1 \end{matrix}$$

These two matrices represent a permutation of size nm: $p_{a_{ij}} = b_{ij}$.

Find the number of cycles in this permutation.

J. Jigsaw Puzzle

Claim: $p_k \equiv k \cdot n \pmod{nm - 1}$.

J. Jigsaw Puzzle

Claim: $p_k \equiv k \cdot n \pmod{nm - 1}$.

Proof: note that $a_{ij} = im + j$ and $b_{ij} = jn + i$.
Then $p_{im+j} = jn + i \equiv jn + inm \pmod{nm - 1} = (im + j) \cdot n$.

J. Jigsaw Puzzle

Claim: $p_k \equiv k \cdot n \pmod{nm - 1}$.

Proof: note that $a_{ij} = im + j$ and $b_{ij} = jn + i$.
Then $p_{im+j} = jn + i \equiv jn + inm \pmod{nm - 1} = (im + j) \cdot n$.

Therefore, except for $p_{nm-1} = nm - 1$, we have
$p_k = k \cdot n \bmod (nm - 1)$ for $k \in [0; nm - 1)$.

J. Jigsaw Puzzle

Let x_k be the smallest positive integer such that $k \cdot n^{x_k} \equiv k \pmod{nm-1}$. Then x_k is the length of the cycle k belongs to.

J. Jigsaw Puzzle

Let x_k be the smallest positive integer such that $k \cdot n^{x_k} \equiv k \pmod{nm-1}$. Then x_k is the length of the cycle k belongs to.

Note that $gcd(n, nm-1) = 1$, and x_k always exists.

J. Jigsaw Puzzle

Let x_k be the smallest positive integer such that $k \cdot n^{x_k} \equiv k \pmod{nm-1}$. Then x_k is the length of the cycle k belongs to.

Note that $gcd(n, nm-1) = 1$, and x_k always exists.

Euler's theorem: $n^{\varphi(nm-1)} \equiv 1 \pmod{nm-1}$.

It follows that x_k divides $\varphi(nm-1)$ for any k.

J. Jigsaw Puzzle

Let x_k be the smallest positive integer such that $k \cdot n^{x_k} \equiv k \pmod{nm-1}$. Then x_k is the length of the cycle k belongs to.

Note that $gcd(n, nm-1) = 1$, and x_k always exists.

Euler's theorem: $n^{\varphi(nm-1)} \equiv 1 \pmod{nm-1}$.

It follows that x_k divides $\varphi(nm-1)$ for any k.

Furthermore, note that x_k only depends on $gcd(k, nm-1)$.

J. Jigsaw Puzzle

Let x_k be the smallest positive integer such that $k \cdot n^{x_k} \equiv k \pmod{nm-1}$. Then x_k is the length of the cycle k belongs to.

Note that $gcd(n, nm-1) = 1$, and x_k always exists.

Euler's theorem: $n^{\varphi(nm-1)} \equiv 1 \pmod{nm-1}$.

It follows that x_k divides $\varphi(nm-1)$ for any k.

Furthermore, note that x_k only depends on $gcd(k, nm-1)$.

Let's fix g, a divisor of $nm-1$. There are $\varphi(\frac{nm-1}{g})$ values of k such that $g = gcd(k, nm-1)$.

We can find x_g by trying all divisors of $\varphi(nm-1)$ and add $\varphi(\frac{nm-1}{g})/x_g$ to the answer.

K. Kingdom Division

Divide an $n \times m$ grid into the maximum number of parts of distinct sizes formed by connected sets of unit squares. Present an example of such a division using characters from $\{\mathtt{A}, \ldots, \mathtt{Z}\}$ to denote the resulting parts.

K. Kingdom Division

A solution to this problem can be obtained in three natural steps.

K. Kingdom Division

A solution to this problem can be obtained in three natural steps.

In the first step we forget for a moment about the requirement of connectivity and ask what sizes of parts would imply the maximum result.

K. Kingdom Division

A solution to this problem can be obtained in three natural steps.

In the first step we forget for a moment about the requirement of connectivity and ask what sizes of parts would imply the maximum result.

The answer is simple, the sizes should be equal to 1, 2, 3, ...

If the last part is smaller than required, we join it with the next-to-last part.

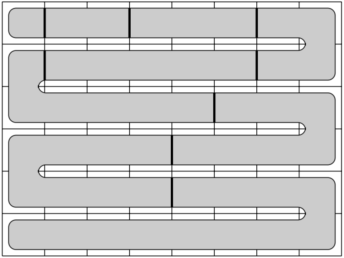
K. Kingdom Division

In the second step we note that the division proposed in the previous step is actually possible to obtain.

K. Kingdom Division

In the second step we note that the division proposed in the previous step is actually possible to obtain.

We can cover the whole grid with a snake-like pattern and then cut the snake at appropriate positions to form the respective parts which will certainly be connected sets.



A	B	B	A	A	A	B	B
D	C	C	C	C	C	B	B
D	D	D	D	D	E	E	E
A	A	A	A	E	E	E	E
A	A	A	A	C	C	C	C
C	C	C	C	C	C	C	C

K. Kingdom Division

The third step is to label parts with the letters from $\{\mathtt{A}, \ldots, \mathtt{Z}\}$ so that no two adjacent parts receive the same label.

K. Kingdom Division

The third step is to label parts with the letters from $\{\mathtt{A}, \dots, \mathtt{Z}\}$ so that no two adjacent parts receive the same label.

A tempting idea would be to label them A, B, C, . . . , Z, A, B, . . . in the order they appear in the snake. However, such a labeling does not work in some cases.

K. Kingdom Division

The third step is to label parts with the letters from $\{\mathtt{A}, \ldots, \mathtt{Z}\}$ so that no two adjacent parts receive the same label.

A tempting idea would be to label them A, B, C, ..., Z, A, B, ... in the order they appear in the snake. However, such a labeling does not work in some cases.

One possible correct solution is to label each part with the smallest letter that does not occur among its neighbours.

K. Kingdom Division

The third step is to label parts with the letters from $\{\mathtt{A}, \dots, \mathtt{Z}\}$ so that no two adjacent parts receive the same label.

A tempting idea would be to label them A, B, C, . . . , Z, A, B, . . . in the order they appear in the snake. However, such a labeling does not work in some cases.

One possible correct solution is to label each part with the smallest letter that does not occur among its neighbours.

Another idea is to label the parts in the first row using the letters A and B, in the second row use the letters C and D, in the third row use the letters E and F, in the fourth row again A and B and so on. Starting from the point when each part covers at least one row, we can stick to just two letters, A and B. This labeling uses only 6 distinct letters.

L. Lovely Numbers

The number N is called lovely if $\frac{\sigma(N)}{N} = \frac{A}{B}$ where $\sigma(N)$ is the sum of all divisors of N.

For given A and B find all lovely numbers between 1 and 10^{14}, inclusive.

L. Lovely Numbers

Consider the prime factorization of N:

$$N = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_k^{e_k}$$

$$\sigma(N) = \frac{p_1^{e_1+1} - 1}{p_1 - 1} \cdot \frac{p_2^{e_2+1} - 1}{p_2 - 1} \cdot \ldots \cdot \frac{p_k^{e_k+1} - 1}{p_k - 1}$$

We can add prime factors of N one by one and recalculate $\sigma(N)$.

L. Lovely Numbers

The solution is brute-force search with pruning.

L. Lovely Numbers

The solution is brute-force search with pruning.

Consider a recursive function $F(n, s, i)$:

- n is the current number;
- s is the sum of divisors of n;
- i is the index of the next prime to be used.

L. Lovely Numbers

The solution is brute-force search with pruning.

Consider a recursive function $F(n, s, i)$:

- n is the current number;
- s is the sum of divisors of n;
- i is the index of the next prime to be used.

For transition, loop over the power k of $prime_i$ and call $F(n \cdot prime_i^k, s \cdot \frac{prime_i^{k+1}-1}{prime_i-1}, i+1)$.

L. Lovely Numbers

Let $S(i)$ be some upper bound of $\frac{sigma(K)}{K}$ for $K \leqslant 10^{14}$ such that K doesn't contain $prime_1, prime_2, \ldots, prime_{i-1}$ in its factorization.

L. Lovely Numbers

Let $S(i)$ be some upper bound of $\frac{sigma(K)}{K}$ for $K \leqslant 10^{14}$ such that K doesn't contain $prime_1, prime_2, \ldots, prime_{i-1}$ in its factorization.

For example, we can find $S(i)$ as follows.

Consider all pairs of the form $(p, \frac{p^{e+1}-1}{p^{e+1}-p})$ for all primes $p \geqslant prime_i$ and $e = 1, 2, \ldots$

If we pick p for the e-th time in our factorization, then the first element of a pair is the number by which we have to multiply n, and the second element of a pair is the number by which we have to multiply s.

L. Lovely Numbers

Let $S(i)$ be some upper bound of $\frac{sigma(K)}{K}$ for $K \leqslant 10^{14}$ such that K doesn't contain $prime_1, prime_2, \ldots, prime_{i-1}$ in its factorization.

For example, we can find $S(i)$ as follows.

Consider all pairs of the form $(p, \frac{p^{e+1}-1}{p^{e+1}-p})$ for all primes $p \geqslant prime_i$ and $e = 1, 2, \ldots$

If we pick p for the e-th time in our factorization, then the first element of a pair is the number by which we have to multiply n, and the second element of a pair is the number by which we have to multiply s.

Consider all pairs in decreasing order of $\frac{second}{first}$. Start with $n = s = 1$, go from the beginning of the list, and keep multiplying n by *first* and s by *second* until $n \geqslant 10^{14}$. After that, $S_i = \frac{s}{n}$.

L. Lovely Numbers

There are several ways we can prune our search:

L. Lovely Numbers

There are several ways we can prune our search:

- If $n \cdot prime_i > 10^{14}$, exit.

L. Lovely Numbers

There are several ways we can prune our search:

- If $n \cdot prime_i > 10^{14}$, exit.
- If $\frac{s}{n} > \frac{A}{B}$, exit.

L. Lovely Numbers

There are several ways we can prune our search:

- If $n \cdot prime_i > 10^{14}$, exit.
- If $\frac{s}{n} > \frac{A}{B}$, exit.
- If $S(i+1) \cdot \frac{s}{n} < \frac{A}{B}$, exit.

L. Lovely Numbers

There are several ways we can prune our search:

- If $n \cdot prime_i > 10^{14}$, exit.
- If $\frac{s}{n} > \frac{A}{B}$, exit.
- If $S(i+1) \cdot \frac{s}{n} < \frac{A}{B}$, exit.
- If $s \cdot B$ contains a prime factor smaller than $prime_i$ which is not a divisor of $n \cdot A$, exit.

L. Lovely Numbers

There are several ways we can prune our search:

- If $n \cdot prime_i > 10^{14}$, exit.
- If $\frac{s}{n} > \frac{A}{B}$, exit.
- If $S(i+1) \cdot \frac{s}{n} < \frac{A}{B}$, exit.
- If $s \cdot B$ contains a prime factor smaller than $prime_i$ which is not a divisor of $n \cdot A$, exit.

These optimizations allow us to find the answers for all values of A and B satisfying the constraints quickly enough.

M. Matryoshkas

We have n dolls, doll i has external volume out_i and internal volume in_i ($in_i < out_i$).

Doll i can be put inside doll j if $out_i < in_j$. If two dolls are located inside the third doll, one of them must be located inside the other.

Find the number of ways to place some dolls inside others so that the total volume of empty space inside all dolls is minimized.

M. Matryoshkas

Let's maintain a set S of dolls that do not contain other dolls yet.

M. Matryoshkas

Let's maintain a set S of dolls that do not contain other dolls yet.

Sort the dolls in decreasing order of out_i. For each doll i, we can either put it inside some doll j from S or not. If we put doll i inside doll j, doll j is removed from S.

M. Matryoshkas

Let's maintain a set S of dolls that do not contain other dolls yet.

Sort the dolls in decreasing order of out_i. For each doll i, we can either put it inside some doll j from S or not. If we put doll i inside doll j, doll j is removed from S.

Note that it is always optimal to put doll i inside doll j if we can: if doll j eventually contains another doll k, by swapping dolls i and k we can decrease the volume of empty space (since $out_i \geqslant out_k$).

M. Matryoshkas

Let's maintain a set S of dolls that do not contain other dolls yet.

Sort the dolls in decreasing order of out_i. For each doll i, we can either put it inside some doll j from S or not. If we put doll i inside doll j, doll j is removed from S.

Note that it is always optimal to put doll i inside doll j if we can: if doll j eventually contains another doll k, by swapping dolls i and k we can decrease the volume of empty space (since $out_i \geqslant out_k$).

Also note that it doesn't matter which doll from S we choose: we will be able to put any following doll k into any doll we can put doll i into (again, since $out_i \geqslant out_k$).

M. Matryoshkas

We have described one optimal way of placing dolls. How to count the number of optimal ways at the same time?

- Initially, the number of ways is 1.

M. Matryoshkas

We have described one optimal way of placing dolls. How to count the number of optimal ways at the same time?

- Initially, the number of ways is 1.
- Whenever a doll can be put inside t dolls from S, the number of ways is multiplied by t. The value of t can be found, for example, with segment tree.

M. Matryoshkas

We have described one optimal way of placing dolls. How to count the number of optimal ways at the same time?

- Initially, the number of ways is 1.
- Whenever a doll can be put inside t dolls from S, the number of ways is multiplied by t. The value of t can be found, for example, with segment tree.
- All dolls with equal external volumes must be processed at once. Suppose there are p such dolls at some stage, and suppose there are t dolls in S that can contain them. If $p > t$, the number of ways is multiplied by $\binom{p}{t}$.

M. Matryoshkas

In fact, no complex data structures are needed.

Instead of set S, it is enough to store just its size t.

The whole algorithm can be summarized as follows:

M. Matryoshkas

In fact, no complex data structures are needed.

Instead of set S, it is enough to store just its size t.

The whole algorithm can be summarized as follows:

- Initially, $t = 0$ and $ans = 1$.

M. Matryoshkas

In fact, no complex data structures are needed.

Instead of set S, it is enough to store just its size t.

The whole algorithm can be summarized as follows:

- Initially, $t = 0$ and $ans = 1$.
- Sort all external and internal volumes V in decreasing order.

M. Matryoshkas

In fact, no complex data structures are needed.

Instead of set S, it is enough to store just its size t.

The whole algorithm can be summarized as follows:

- Initially, $t = 0$ and $ans = 1$.
- Sort all external and internal volumes V in decreasing order.
- If there are k dolls with external volume V, multiply ans by $\binom{\max(k,t)}{\min(k,t)} \cdot (\min(k,t))!$ and set t to $max(t-k, 0)$.

M. Matryoshkas

In fact, no complex data structures are needed.

Instead of set S, it is enough to store just its size t.

The whole algorithm can be summarized as follows:

- Initially, $t = 0$ and $ans = 1$.
- Sort all external and internal volumes V in decreasing order.
- If there are k dolls with external volume V, multiply ans by $\binom{\max(k,t)}{\min(k,t)} \cdot (\min(k,t))!$ and set t to $max(t-k,0)$.
- If there are k dolls with internal volume V, increase t by k.

In the end, ans contains the answer. The complexity is $\mathcal{O}(n \log n)$.