Warsaw U Contest Editorial

November 14th, 2016

by Mikhail Tikhomirov (MIPT)

Moscow ACM ICPC Workshop, MIPT, 2016

A. Arkanoid

There are k 1×1 square obstacles on a rectangular $n \times m$ field. A ball starts moving diagonally to axes from a certain point. The ball reflects from the walls. When the ball collides with an obstacle, the latter is destroyed and the ball reflects naturally. Determine the time when the last obstacle is destroyed.

A. Arkanoid

There are k 1×1 square obstacles on a rectangular $n \times m$ field. A ball starts moving diagonally to axes from a certain point. The ball reflects from the walls. When the ball collides with an obstacle, the latter is destroyed and the ball reflects naturally. Determine the time when the last obstacle is destroyed.

Outline: modelling with effective finding of the next obstacle to break.

A. Arkanoid

A naive simulating approach can work in $O(nmk)$ time in the worst case.

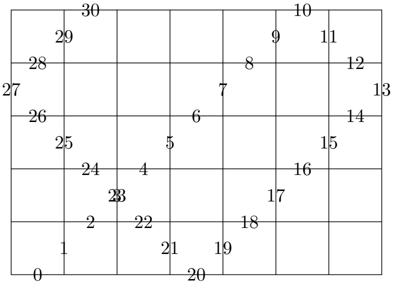
A. Arkanoid

A naive simulating approach can work in $O(nmk)$ time in the worst case.

The hardest part is to find for a certain ball position and a set of obstacles (some of the initial ones could get destroyed) which obstacle is the next to be destroyed.

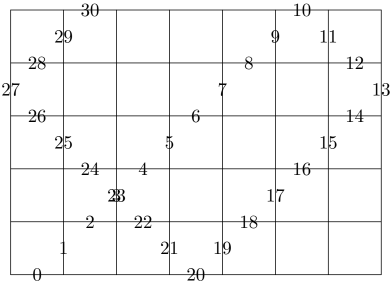
A. Arkanoid

Let us number all possible pairs of position and direction of the ball starting from a certain point.



A. Arkanoid

Let us number all possible pairs of position and direction of the ball starting from a certain point.



If one has the number k of the situation (position + direction), one can transform it to actual position and velocity coordinates. Indeed, consider time moment k. If the ball started from $(0.5, 0)$ moving to the upper right, it must have made $\lfloor (0.5k + 0.5)/m \rfloor$ bounces from vertical sides and $\lfloor 0.5k/n \rfloor$ from horizontal sides. This information is enough to find the coordinates and velocity.

A. Arkanoid

To answer a converse question, that is, determine the number k from coordinates and velocity, first notice that the pair (x-coordinate, x-velocity) periodically repeats with period $4m$; the same holds for y and period $4n$. Thus, one has to solve a system of modular equations of sort:

$$k \equiv k_x \pmod{4m}, k \equiv k_y \pmod{4n},$$

where numbers k_x and k_y correspond to the position and direction.

A. Arkanoid

To answer a converse question, that is, determine the number k from coordinates and velocity, first notice that the pair (x-coordinate, x-velocity) periodically repeats with period $4m$; the same holds for y and period $4n$. Thus, one has to solve a system of modular equations of sort:

$$k \equiv k_x \pmod{4m}, k \equiv k_y \pmod{4n},$$

where numbers k_x and k_y correspond to the position and direction.

One way of solving a system of similar form is to use the Chinese remainder theorem.

A. Arkanoid

Now consider a particular obstacle. There are 8 different pairs (position, velocity) that correspond to hitting the obstacle from different angles. Each of these pairs has a unique number (if we use the situations indexing described above). We will store all these numbers in a data structure that supports finding lower/upper bound.

A. Arkanoid

Now consider a particular obstacle. There are 8 different pairs (position, velocity) that correspond to hitting the obstacle from different angles. Each of these pairs has a unique number (if we use the situations indexing described above). We will store all these numbers in a data structure that supports finding lower/upper bound.

Suppose that we are currently in a situation with index k. The current direction can correspond to increasing or decreasing k over time. Now, finding the next collision can be done with certain lower/upper bound query to the data structure.

A. Arkanoid

Now consider a particular obstacle. There are 8 different pairs (position, velocity) that correspond to hitting the obstacle from different angles. Each of these pairs has a unique number (if we use the situations indexing described above). We will store all these numbers in a data structure that supports finding lower/upper bound.

Suppose that we are currently in a situation with index k. The current direction can correspond to increasing or decreasing k over time. Now, finding the next collision can be done with certain lower/upper bound query to the data structure.

Having found the next collision, we should erase all entries that correspond to the recently destroyed obstacle. We proceed until all obstacles are destroyed.

A. Arkanoid

Now consider a particular obstacle. There are 8 different pairs (position, velocity) that correspond to hitting the obstacle from different angles. Each of these pairs has a unique number (if we use the situations indexing described above). We will store all these numbers in a data structure that supports finding lower/upper bound.

Suppose that we are currently in a situation with index k. The current direction can correspond to increasing or decreasing k over time. Now, finding the next collision can be done with certain lower/upper bound query to the data structure.

Having found the next collision, we should erase all entries that correspond to the recently destroyed obstacle. We proceed until all obstacles are destroyed.

The solution has $O(k \log k)$ complexity.

B. Vari-directional Streets

A vertex v of a directed graph is *good* if for every vertex u either u is reachable from v or v is reachable from u. Find the set of good vertices of a given digraph.

B. Vari-directional Streets

A vertex v of a directed graph is *good* if for every vertex u either u is reachable from v or v is reachable from u. Find the set of good vertices of a given digraph.

Outline: condense the strongly connected components of the digraph, obtain a simple criterion for a DAG using topsort properties.

B. Vari-directional Streets

DAG case

First, suppose that a given digraph is a DAG (*directed acyclic graph*). Can we determine the set of good vertices in this case?

B. Vari-directional Streets

DAG case

First, suppose that a given digraph is a DAG (*directed acyclic graph*). Can we determine the set of good vertices in this case?

Choose a topological ordering of the DAG arbitrarily. For simplicity we will identify vertices of the DAG with their indices in topsort.

B. Vari-directional Streets

DAG case

First, suppose that a given digraph is a DAG (*directed acyclic graph*). Can we determine the set of good vertices in this case?

Choose a topological ordering of the DAG arbitrarily. For simplicity we will identify vertices of the DAG with their indices in topsort.

A vertex v is good iff all vertices $u > v$ are reachable from v, and v is reachable from any $u < v$. We will check the first condition for all vertices; the second one can be checked in a completely symmetrical way.

B. Vari-directional Streets

DAG case

For a vertex $u > v$ let $deg_{+}(v|u)$ denote the number of edges (w, u) with $w \geqslant v$. Let us call a vertex u a *v-source* if $deg_{+}(v|u) = 0$ (that is, u is a source in the part of the graph to the right of v including v).

B. Vari-directional Streets

DAG case

For a vertex $u > v$ let $deg_{+}(v|u)$ denote the number of edges (w, u) with $w \geqslant v$. Let us call a vertex u a *v-source* if $deg_{+}(v|u) = 0$ (that is, u is a source in the part of the graph to the right of v including v).

Note that existence of a vertex $u > v$ that is unreachable from v is equivalent to existence of a v-source different from v.

B. Vari-directional Streets

DAG case

For a vertex $u > v$ let $deg_+(v|u)$ denote the number of edges (w, u) with $w \geqslant v$. Let us call a vertex u a *v-source* if $deg_+(v|u) = 0$ (that is, u is a source in the part of the graph to the right of v including v).

Note that existence of a vertex $u > v$ that is unreachable from v is equivalent to existence of a v-source different from v.

Let us iterate over all possible v from left to right, and maintain $deg_+(v|u)$ for all $u \geqslant v$ along with the number of vertices with $deg_+(v|u) = 0$.

B. Vari-directional Streets

DAG case

For a vertex $u > v$ let $deg_+(v|u)$ denote the number of edges (w, u) with $w \geqslant v$. Let us call a vertex u a *v-source* if $deg_+(v|u) = 0$ (that is, u is a source in the part of the graph to the right of v including v).

Note that existence of a vertex $u > v$ that is unreachable from v is equivalent to existence of a v-source different from v.

Let us iterate over all possible v from left to right, and maintain $deg_+(v|u)$ for all $u \geqslant v$ along with the number of vertices with $deg_+(v|u) = 0$.

To move from v to $v + 1$ simply decrease in-degree of all vertices directly reachable from v.

B. Vari-directional Streets

DAG case

For a vertex $u > v$ let $deg_+(v|u)$ denote the number of edges (w, u) with $w \geqslant v$. Let us call a vertex u a *v-source* if $deg_+(v|u) = 0$ (that is, u is a source in the part of the graph to the right of v including v).

Note that existence of a vertex $u > v$ that is unreachable from v is equivalent to existence of a v-source different from v.

Let us iterate over all possible v from left to right, and maintain $deg_+(v|u)$ for all $u \geqslant v$ along with the number of vertices with $deg_+(v|u) = 0$.

To move from v to $v + 1$ simply decrease in-degree of all vertices directly reachable from v.

To check the symmetrical condition consider the reversed graph. A vertex v is good if doesn't have a v-source in both cases.

B. Vari-directional Streets

To obtain the solution for a general digraph note that all vertices in the same SCC (*strongly connected components*) either are all good or all bad.

B. Vari-directional Streets

To obtain the solution for a general digraph note that all vertices in the same SCC (*strongly connected components*) either are all good or all bad.

Build all SCC's and the condensation of the digraph (compressed graph with vertices in SCC's and edges between different SCC's). Apply the DAG solution to the condensation, output all vertices in good SCC's.

B. Vari-directional Streets

To obtain the solution for a general digraph note that all vertices in the same SCC (*strongly connected components*) either are all good or all bad.

Build all SCC's and the condensation of the digraph (compressed graph with vertices in SCC's and edges between different SCC's). Apply the DAG solution to the condensation, output all vertices in good SCC's.

Both condensation construction and DAG case are solvable in $O(n+m)$ time.

C. Club members

Find a hamiltonian cycle in an cube graph with 2^n vertices that contains a given perfect matching.

C. Club members

Find a hamiltonian cycle in an cube graph with 2^n vertices that contains a given perfect matching.

Outline: constructive solution that reduces to smaller problems.

C. Club members

Find a hamiltonian cycle in an cube graph with 2^n vertices that contains a given perfect matching.

Outline: constructive solution that reduces to smaller problems.

For details read the enclosed solution.

D. Necklace

We are given an array of n integers. All subsequences of the array are ordered by sum of the elements; subsequences with equal sum are ordered lexicographically as sorted tuples of indices. Find k-th subsequence in this ordering. $n, k \leqslant 10^6$.

D. Necklace

We are given an array of n integers. All subsequences of the array are ordered by sum of the elements; subsequences with equal sum are ordered lexicographically as sorted tuples of indices. Find k-th subsequence in this ordering. $n, k \leqslant 10^6$.

Outline: a Dijkstra-like approach with enough optimizations.

D. Necklace

Let us generate all subsequences in order. We start from the empty sequence. At each moment we will have a data structure of all candidates to be the next minimum. A general idea is to extract minimum from the structure and try all options to expand it by an element it doesn't contain yet.

D. Necklace

Let us generate all subsequences in order. We start from the empty sequence. At each moment we will have a data structure of all candidates to be the next minimum. A general idea is to extract minimum from the structure and try all options to expand it by an element it doesn't contain yet.

We have to take care not to add the same entry to the structure twice. One way of doing it is to only append the elements that go later in the sequence than the previous last element, hence there is a unique order of adding elements for each subsequence.

D. Necklace

Let us generate all subsequences in order. We start from the empty sequence. At each moment we will have a data structure of all candidates to be the next minimum. A general idea is to extract minimum from the structure and try all options to expand it by an element it doesn't contain yet.

We have to take care not to add the same entry to the structure twice. One way of doing it is to only append the elements that go later in the sequence than the previous last element, hence there is a unique order of adding elements for each subsequence.

Optimization 1: if the size of the structure is greater than k, we can remove the largest entry.

D. Necklace

Let us generate all subsequences in order. We start from the empty sequence. At each moment we will have a data structure of all candidates to be the next minimum. A general idea is to extract minimum from the structure and try all options to expand it by an element it doesn't contain yet.

We have to take care not to add the same entry to the structure twice. One way of doing it is to only append the elements that go later in the sequence than the previous last element, hence there is a unique order of adding elements for each subsequence.

Optimization 1: if the size of the structure is greater than k, we can remove the largest entry.

Optimization 2: sort the given numbers (don't forget to store their original indices), stop trying to append a number when the sum becomes too great to fit in the structure.

D. Necklace

This approach might still take too long since we have a lot of options to append a number on each step.

D. Necklace

This approach might still take too long since we have a lot of options to append a number on each step.

Optimization 3: leave only $O(1)$ transitions from each state by introducting new information. Instead of (sum, subset of indices, [possibly lower bound for index]), we will now have (lower bound for sum after adding the number i, subset of indices, current position i).

D. Necklace

This approach might still take too long since we have a lot of options to append a number on each step.

Optimization 3: leave only $O(1)$ transitions from each state by introducting new information. Instead of (sum, subset of indices, [possibly lower bound for index]), we will now have (lower bound for sum after adding the number i, subset of indices, current position i).

While processing the next state we have to try to add number i and add the subset to the list of generated sequences. However, in the sequel we may opt to skip the number. The two transitions are to continue with the number a_i included or not included in the subset; in both cases the current position is increased by 1. Note that the lower bound for the sum does not decrease in any case.

D. Necklace

This approach might still take too long since we have a lot of options to append a number on each step.

Optimization 3: leave only $O(1)$ transitions from each state by introducting new information. Instead of (sum, subset of indices, [possibly lower bound for index]), we will now have (lower bound for sum after adding the number i, subset of indices, current position i).

While processing the next state we have to try to add number i and add the subset to the list of generated sequences. However, in the sequel we may opt to skip the number. The two transitions are to continue with the number a_i included or not included in the subset; in both cases the current position is increased by 1. Note that the lower bound for the sum does not decrease in any case.

It is easy to see that the target sequence will not have more than $\log_2 k$ elements since all the subsets of a subsequence precede it in the order. Thus the described solution has complexity $O(n \log n + k \log^2 k)$, since we compare two states in $O(\log_2 k)$ time.

E. Amusing Journeys

We are given a connected graph without multiple edges. Determine if all simple closed paths in the graph have the same length. If that is the case, count these cycles modulo $10^9 + 7$.

E. Amusing Journeys

We are given a connected graph without multiple edges. Determine if all simple closed paths in the graph have the same length. If that is the case, count these cycles modulo $10^9 + 7$.

Outline: if the condition holds, the biconnected blocks have very special form.

E. Amusing Journeys

Each cycle of the graph lies completely inside of a biconnected component. Consider a separate component.

E. Amusing Journeys

Each cycle of the graph lies completely inside of a biconnected component. Consider a separate component.

If the component consists of a single edge, then it contains no cycles.

E. Amusing Journeys

Each cycle of the graph lies completely inside of a biconnected component. Consider a separate component.

If the component consists of a single edge, then it contains no cycles.

If the component is a cycle of length l, then we have $2l$ closed paths for every choice of starting point and direction.

E. Amusing Journeys

Each cycle of the graph lies completely inside of a biconnected component. Consider a separate component.

If the component consists of a single edge, then it contains no cycles.

If the component is a cycle of length l, then we have $2l$ closed paths for every choice of starting point and direction.

Otherwise, suppose that all simple cycles contain l edges each. Consider a pair C_1, C_2 of intersecting cycles, and take their symmetrical difference S (that is, the set of edges that are present in exactly one of the cycles).

E. Amusing Journeys

Each cycle of the graph lies completely inside of a biconnected component. Consider a separate component.

If the component consists of a single edge, then it contains no cycles.

If the component is a cycle of length l, then we have $2l$ closed paths for every choice of starting point and direction.

Otherwise, suppose that all simple cycles contain l edges each. Consider a pair C_1, C_2 of intersecting cycles, and take their symmetrical difference S (that is, the set of edges that are present in exactly one of the cycles).

The set S has less than $2l$ edges, and can be decomposed into simple cycles. By assumption, S must itself be a cycle of length l. If that is the case, the intersection of C_1 and C_2 must be a path of length $l/2$. Thus $C_1 \cup C_2$ is a graph that consists of three edge-disjoint paths of length $l/2$ between a pair of vertices v and u.

E. Amusing Journeys

Each cycle of the graph lies completely inside of a biconnected component. Consider a separate component.

If the component consists of a single edge, then it contains no cycles.

If the component is a cycle of length l, then we have $2l$ closed paths for every choice of starting point and direction.

Otherwise, suppose that all simple cycles contain l edges each. Consider a pair C_1, C_2 of intersecting cycles, and take their symmetrical difference S (that is, the set of edges that are present in exactly one of the cycles).

The set S has less than $2l$ edges, and can be decomposed into simple cycles. By assumption, S must itself be a cycle of length l. If that is the case, the intersection of C_1 and C_2 must be a path of length $l/2$. Thus $C_1 \cup C_2$ is a graph that consists of three edge-disjoint paths of length $l/2$ between a pair of vertices v and u.

Note that no additional edges can be added between vertices of $C_1 \cup C_2$ so that each cycle has length l.

E. Amusing Journeys

If $C_1 \cup C_2$ is not the whole component yet, there must be a cycle that has common edges with C_1 or C_2, but doesn't lie completely inside $C_1 \cup C_2$. By a similar argument, the new cycle must consist of two paths of length $l/2$ between v and u: one old and one new.

E. Amusing Journeys

If $C_1 \cup C_2$ is not the whole component yet, there must be a cycle that has common edges with C_1 or C_2, but doesn't lie completely inside $C_1 \cup C_2$. By a similar argument, the new cycle must consist of two paths of length $l/2$ between v and u: one old and one new.

It follows that all simple cycles in a biconnected component have same length iff the component consists of several paths of equal length between a certain pair of vertices.

E. Amusing Journeys

If $C_1 \cup C_2$ is not the whole component yet, there must be a cycle that has common edges with C_1 or C_2, but doesn't lie completely inside $C_1 \cup C_2$. By a similar argument, the new cycle must consist of two paths of length $l/2$ between v and u: one old and one new.

It follows that all simple cycles in a biconnected component have same length iff the component consists of several paths of equal length between a certain pair of vertices.

In case the component is not an edge nor a cycle, these two vertices should be exactly the vertices with degree greater than 2.

E. Amusing Journeys

If $C_1 \cup C_2$ is not the whole component yet, there must be a cycle that has common edges with C_1 or C_2, but doesn't lie completely inside $C_1 \cup C_2$. By a similar argument, the new cycle must consist of two paths of length $l/2$ between v and u: one old and one new.

It follows that all simple cycles in a biconnected component have same length iff the component consists of several paths of equal length between a certain pair of vertices.

In case the component is not an edge nor a cycle, these two vertices should be exactly the vertices with degree greater than 2.

The path structure can be easily found if the component is given.

E. Amusing Journeys

If $C_1 \cup C_2$ is not the whole component yet, there must be a cycle that has common edges with C_1 or C_2, but doesn't lie completely inside $C_1 \cup C_2$. By a similar argument, the new cycle must consist of two paths of length $l/2$ between v and u: one old and one new.

It follows that all simple cycles in a biconnected component have same length iff the component consists of several paths of equal length between a certain pair of vertices.

In case the component is not an edge nor a cycle, these two vertices should be exactly the vertices with degree greater than 2.

The path structure can be easily found if the component is given.

Finally, we can decompose the graph into biconnected components in $O(n+m)$ time. We should also check that the cycle lengths for different components are equal.

F. Nim with a twist

Count the number of ways to remove kd Nim heaps out of n so that the second player wins. $d \leqslant 10$, total size of the heaps $\leqslant 10^7$.

F. Nim with a twist

Count the number of ways to remove kd Nim heaps out of n so that the second player wins. $d \leqslant 10$, total size of the heaps $\leqslant 10^7$.

Outline: standard DP with optimization.

F. Nim with a twist

Fact

The second player wins iff XOR of all heap sizes is zero.

F. Nim with a twist

Fact

The second player wins iff XOR of all heap sizes is zero.

Note that XOR of heap sizes a_i is less than $2 \max a_i$.

F. Nim with a twist

Fact

The second player wins iff XOR of all heap sizes is zero.

Note that XOR of heap sizes a_i is less than $2 \max a_i$.

An $O(nd \max a_i)$ solution: $dp_{i,m,x}$ = number of subsets among first i heaps such that the number of omitted heaps is m modulo d, and XOR of all taken heaps' sizes if x.

This DP has $O(nd \max a_i)$ states and transitions.

F. Nim with a twist

Fact

The second player wins iff XOR of all heap sizes is zero.

Note that XOR of heap sizes a_i is less than $2\max a_i$.

An $O(nd\max a_i)$ solution: $dp_{i,m,x}$ = number of subsets among first i heaps such that the number of omitted heaps is m modulo d, and XOR of all taken heaps' sizes if x.

This DP has $O(nd\max a_i)$ states and transitions.

Optimization: let us process a_i by increasing. By the time we process a_i, we can't get XOR of some smaller numbers greater than $2a_i$, so we won't store such values. Now appending a single number a_i is done in $O(a_i)$, for the total complexity $O(d\sum a_i + n\log n)$.

G. Parade

In a given tree find a simple path such that the number of edges with exactly one endpoint inside the path is maximized.

G. Parade

In a given tree find a simple path such that the number of edges with exactly one endpoint inside the path is maximized.

Outline: standard subtree DP.

G. Parade

First we solve a “vertical” version of the problem. Make the tree rooted, and let *down*(v) be the answer if the path goes from v into its subtree, and we only consider the edges in the subtree (no edge from v to the parent). *down*(v) will allow for a single-vertex path (unlike the original problem).

G. Parade

First we solve a “vertical” version of the problem. Make the tree rooted, and let $down(v)$ be the answer if the path goes from v into its subtree, and we only consider the edges in the subtree (no edge from v to the parent). $down(v)$ will allow for a single-vertex path (unlike the original problem).

If ch_v is the number of children of v, then

$$down(v) = \max\left(ch(v), ch(v) - 1 + \max_{u \text{ is a child of } v} down(u)\right).$$

G. Parade

Now let the path be not necessarily vertical, and let v be the highest vertex in the path.

G. Parade

Now let the path be not necessarily vertical, and let v be the highest vertex in the path.

If v is the root of the tree, then the maximal length of such path is either

$$ch(v) - 1 + \max_{u \text{ is a child of } v} down(u)$$

if v is an endpoint,

G. Parade

Now let the path be not necessarily vertical, and let v be the highest vertex in the path.

If v is the root of the tree, then the maximal length of such path is either

$$ch(v) - 1 + \max_{u \text{ is a child of } v} down(u)$$

if v is an endpoint,

or

$$ch(v) - 2 + \max_{u_1, u_2 \text{ — different children of } v} (down(u_1) + down(u_2))$$

if v is a proper LCA of endpoints.

G. Parade

Now let the path be not necessarily vertical, and let v be the highest vertex in the path.

If v is the root of the tree, then the maximal length of such path is either

$$ch(v) - 1 + \max_{u \text{ is a child of } v} down(u)$$

if v is an endpoint,

or

$$ch(v) - 2 + \max_{u_1, u_2 \text{ — different children of } v} (down(u_1) + down(u_2))$$

if v is a proper LCA of endpoints.

If v is not the root, we have to add 1 to both values to account for the parent edge.

G. Parade

Now let the path be not necessarily vertical, and let v be the highest vertex in the path.

If v is the root of the tree, then the maximal length of such path is either

$$ch(v) - 1 + \max_{u \text{ is a child of } v} down(u)$$

if v is an endpoint,

or

$$ch(v) - 2 + \max_{u_1, u_2 \text{ — different children of } v} (down(u_1) + down(u_2))$$

if v is a proper LCA of endpoints.

If v is not the root, we have to add 1 to both values to account for the parent edge.

All the above can be computed in $O(n)$ time.

H. Messenger

In a directed graph, count the number of paths of length l from v to u that don't pass through v and u other than at start and finish. Many queries of v, u, l.

H. Messenger

In a directed graph, count the number of paths of length l from v to u that don't pass through v and u other than at start and finish. Many queries of v, u, l.

Outline: count the standard DP $paths_{v,u,l}$ for the number of all paths without any constraints, then carefully subtract all excess paths.

H. Messenger

We will have to count several DP's. First is the standard $paths_{v,u,l}$ for the total number of paths of length l from v to u:

$$paths_{v,v,0} = 1$$

$$paths_{v,u,l} = \sum_{w:(v,w)\text{ — an edge}} paths_{w,u,l-1}$$

H. Messenger

We will have to count several DP's. First is the standard $paths_{v,u,l}$ for the total number of paths of length l from v to u:

$$paths_{v,v,0} = 1$$

$$paths_{v,u,l} = \sum_{w:(v,w) \text{ — an edge}} paths_{w,u,l-1}$$

$paths'_{v,u,l}$ — the number of paths of length l from v to u that contain v only as the start. We want to subtract all non-suitable paths from total. Let l' be the last moment a bad path passes through v. Hence the formula:

$$paths'_{v,u,l} = paths_{v,u,l} - \sum_{l'=1}^{l-1} paths_{v,v,l'} paths'_{v,u,l-l'}$$

H. Messenger

$cycle'_{v,u,l}$ — the number of paths of length l from v to v avoiding u. Let l' be the last moment a bad path passes through u. Then

$$cycle'_{v,u,l} = dp_{v,v,l} - \sum_{l'=1}^{l-1} dp_{v,u,l'}\, paths'_{u,v,l-l'}$$

H. Messenger

$cycle'_{v,u,l}$ — the number of paths of length l from v to v avoiding u. Let l' be the last moment a bad path passes through u. Then

$$cycle'_{v,u,l} = dp_{v,v,l} - \sum_{l'=1}^{l-1} dp_{v,u,l'} \, paths'_{u,v,l-l'}$$

Finally, $paths''_{v,u,l}$ is the answer to the original problem. In a similar way we have

$$paths''_{v,u,l} = paths'_{v,u,l} - \sum_{l'=1}^{l-1} paths'_{v,u,l'} \, cycle'_{u,v,l-l'}$$

H. Messenger

$cycle'_{v,u,l}$ — the number of paths of length l from v to v avoiding u. Let l' be the last moment a bad path passes through u. Then

$$cycle'_{v,u,l} = dp_{v,v,l} - \sum_{l'=1}^{l-1} dp_{v,u,l'} paths'_{u,v,l-l'}$$

Finally, $paths''_{v,u,l}$ is the answer to the original problem. In a similar way we have

$$paths''_{v,u,l} = paths'_{v,u,l} - \sum_{l'=1}^{l-1} paths'_{v,u,l'} cycle'_{u,v,l-l'}$$

All the above values can be computed in $O(nml + n^2 l^2)$ time, and each query can be answered in $O(1)$ time.

I. Diligent Johny

We are given a permutation. We repeatedly go from the current permutation to lexicographically previous one using minimal number of element swaps (not necessarily adjacent!). How many swaps we will make in total before we arrive to the $(1, \ldots, n)$ permutation?

I. Diligent Johny

We are given a permutation. We repeatedly go from the current permutation to lexicographically previous one using minimal number of element swaps (not necessarily adjacent!). How many swaps we will make in total before we arrive to the $(1, \ldots, n)$ permutation?

Outline: combinatorial argument, then "number-by-permutation"-like algorithm with RSQ.

I. Diligent Johny

Note that we can just as well go from $(1, \ldots, n)$ to the next permutation until we arrive at the given one. In the sequel we will use this restatement.

I. Diligent Johny

Note that we can just as well go from $(1, \ldots, n)$ to the next permutation until we arrive at the given one. In the sequel we will use this restatement.

Let $f(n)$ be the total number of swaps to proceed from $(1, \ldots, n)$ to $(n, \ldots, 1)$.

I. Diligent Johny

Note that we can just as well go from $(1, \ldots, n)$ to the next permutation until we arrive at the given one. In the sequel we will use this restatement.

Let $f(n)$ be the total number of swaps to proceed from $(1, \ldots, n)$ to $(n, \ldots, 1)$.

To do that, first we have to get from $(1, \ldots, n)$ to $(1, n, \ldots, 2)$. Next we have to change $(1, n, \ldots, 2)$ into $(2, 1, 3, \ldots, n)$, then to $(2, n, \ldots, 3, 1)$, and so on.

I. Diligent Johny

Note that we can just as well go from $(1, \ldots, n)$ to the next permutation until we arrive at the given one. In the sequel we will use this restatement.

Let $f(n)$ be the total number of swaps to proceed from $(1, \ldots, n)$ to $(n, \ldots, 1)$.

To do that, first we have to get from $(1, \ldots, n)$ to $(1, n, \ldots, 2)$. Next we have to change $(1, n, \ldots, 2)$ into $(2, 1, 3, \ldots, n)$, then to $(2, n, \ldots, 3, 1)$, and so on.

In general, we have n steps of "swap the suffix" sort. Each of them take $f(n-1)$ steps.

I. Diligent Johny

Note that we can just as well go from $(1, \dots, n)$ to the next permutation until we arrive at the given one. In the sequel we will use this restatement.

Let $f(n)$ be the total number of swaps to proceed from $(1, \dots, n)$ to $(n, \dots, 1)$.

To do that, first we have to get from $(1, \dots, n)$ to $(1, n, \dots, 2)$. Next we have to change $(1, n, \dots, 2)$ into $(2, 1, 3, \dots, n)$, then to $(2, n, \dots, 3, 1)$, and so on.

In general, we have n steps of "swap the suffix" sort. Each of them take $f(n-1)$ steps.

Between these steps we have to change $(k, n, \dots, k+1, k-1, \dots, 1)$ to $(k+1, 1, \dots, k, k+2, \dots, n)$, where $k = 1, \dots, n-1$.

I. Diligent Johny

Fact

The minimal number of swaps to change a permutation p into permutation q is equal to n - (number of cycles in $p^{-1}q$).

I. Diligent Johny

Fact

The minimal number of swaps to change a permutation p into permutation q is equal to n - (number of cycles in $p^{-1}q$).

One can check with some case analysis (or with brute-force for small numbers) that the number of swaps needed to change $(k, n, \ldots, k+1, k-1, \ldots, 1)$ into $(k+1, 1, \ldots, k, k+2, \ldots, n)$ doesn't depend on k and is equal to $\lceil n/2 \rceil$.

I. Diligent Johny

Fact

The minimal number of swaps to change a permutation p into permutation q is equal to n - (number of cycles in $p^{-1}q$).

One can check with some case analysis (or with brute-force for small numbers) that the number of swaps needed to change $(k, n, \ldots, k+1, k-1, \ldots, 1)$ into $(k+1, 1, \ldots, k, k+2, \ldots, n)$ doesn't depend on k and is equal to $\lceil n/2 \rceil$.

Thus, we have $f(n) = nf(n-1) + (n-1)\lceil n/2 \rceil$. Values of this recurrence can readily be found in $O(n)$ time.

I. Diligent Johny

Now to solve the “partial” problem: find the number of steps to obtain $(p_1, \ldots, p_n)$ from $(1, \ldots, n)$.

I. Diligent Johny

Now to solve the “partial” problem: find the number of steps to obtain $(p_1, \ldots, p_n)$ from $(1, \ldots, n)$.

Suppose we have just obtained the correct prefix of length $l - 1$, so all the rest elements p_l, . . . , p_n are currently in increasing order.

I. Diligent Johny

Now to solve the "partial" problem: find the number of steps to obtain $(p_1, \ldots, p_n)$ from $(1, \ldots, n)$.

Suppose we have just obtained the correct prefix of length $l - 1$, so all the rest elements p_l, ..., p_n are currently in increasing order.

Let x_l be the index of p_l among the remaining elements if we order them by increasing. To place p_l in l-th position we have to do $x_l - 1$ repetitions of "swap the suffix" and "apply next permutation so that l-th element increases".

I. Diligent Johny

Now to solve the “partial” problem: find the number of steps to obtain $(p_1, \ldots, p_n)$ from $(1, \ldots, n)$.

Suppose we have just obtained the correct prefix of length $l - 1$, so all the rest elements p_l, ..., p_n are currently in increasing order.

Let x_l be the index of p_l among the remaining elements if we order them by increasing. To place p_l in l-th position we have to do $x_l - 1$ repetitions of “swap the suffix” and “apply next permutation so that l-th element increases”.

By previous arguments, we have to perform $(x_l - 1)(f(n - l) + \lceil (n - l)/2 \rceil)$ swaps. After that, p_l is in its place, and all the later elements are sorted, thus we reduce to a smaller problem.

I. Diligent Johny

Now to solve the "partial" problem: find the number of steps to obtain $(p_1, \ldots, p_n)$ from $(1, \ldots, n)$.

Suppose we have just obtained the correct prefix of length $l - 1$, so all the rest elements p_l, …, p_n are currently in increasing order.

Let x_l be the index of p_l among the remaining elements if we order them by increasing. To place p_l in l-th position we have to do $x_l - 1$ repetitions of "swap the suffix" and "apply next permutation so that l-th element increases".

By previous arguments, we have to perform $(x_l - 1)(f(n - l) + \lceil (n - l)/2 \rceil)$ swaps. After that, p_l is in its place, and all the later elements are sorted, thus we reduce to a smaller problem.

Values of x_l can be found using any kind of RSQ data structure in $O(n \log n)$.

J. Not Nim

Two players are playing a game with n pairs of heaps of stones. Initially both heaps in i-th pair contain a_i stones. The first player can remove any number of stones from any heap. The second player must move several stones between heaps in some pair. The first player wants to remove all stones in minimal number of moves, while the second player wants to play as long as possible. Find out the number of moves in the game if both play optimally.

J. Not Nim

Two players are playing a game with n pairs of heaps of stones. Initially both heaps in i-th pair contain a_i stones. The first player can remove any number of stones from any heap. The second player must move several stones between heaps in some pair. The first player wants to remove all stones in minimal number of moves, while the second player wants to play as long as possible. Find out the number of moves in the game if both play optimally.

Outline: evil problem with hard-to-identify cases.

J. Not Nim

Couple of simple observations:

- The first player always empties one of the heaps.

J. Not Nim

Couple of simple observations:

- The first player always empties one of the heaps.
- After the first player emptied a heap, the second player should try to even out the heaps in this pair when this is possible.

J. Not Nim

The first player can easily win in $F = \sum_{i=1}^{n}(2 + \lfloor \log_2 a_i \rfloor)$ (we only count first player's moves here), but sometimes he can win faster.

J. Not Nim

The first player can easily win in $F = \sum_{i=1}^{n}(2 + \lfloor \log_2 a_i \rfloor)$ (we only count first player's moves here), but sometimes he can win faster.

We will say that the first player can *snatch* a move if he forces the second player to move in a situation when each pair of heaps is either empty or has $(2^k, 2^k)$ stones for certain integer k (probably different for different pairs).

J. Not Nim

The first player can easily win in $F = \sum_{i=1}^{n}(2 + \lfloor \log_2 a_i \rfloor)$ (we only count first player's moves here), but sometimes he can win faster.

We will say that the first player can *snatch* a move if he forces the second player to move in a situation when each pair of heaps is either empty or has $(2^k, 2^k)$ stones for certain integer k (probably different for different pairs).

These are the only situations that help the first player to get under the upper bound F. Indeed, if the second player could skip moves, then F would be the exact number of moves. Thus, the only way to do better than F is to force the second player to do harmful moves.

J. Not Nim

The first player can easily win in $F = \sum_{i=1}^{n}(2 + \lfloor \log_2 a_i \rfloor)$ (we only count first player's moves here), but sometimes he can win faster.

We will say that the first player can *snatch* a move if he forces the second player to move in a situation when each pair of heaps is either empty or has $(2^k, 2^k)$ stones for certain integer k (probably different for different pairs).

These are the only situations that help the first player to get under the upper bound F. Indeed, if the second player could skip moves, then F would be the exact number of moves. Thus, the only way to do better than F is to force the second player to do harmful moves.

Note that if some pair contains unequal heaps, than the second player can effectively skip a move, thus such situations are not appealing to the first player.

J. Not Nim

Suppose that the first player will move in a pair that currently contains (x, x) stones. Forcing a second player's move here will result in $(x-1, 0)$ instead of $(x, 0)$ (if the second player could skip). The move will be harmful if $\lfloor \log_2(x-1) \rfloor < \lfloor \log_2 x \rfloor$, or, equivalently, $x = 2^k$.

J. Not Nim

Suppose that the first player will move in a pair that currently contains (x, x) stones. Forcing a second player's move here will result in $(x-1, 0)$ instead of $(x, 0)$ (if the second player could skip). The move will be harmful if $\lfloor \log_2(x-1) \rfloor < \lfloor \log_2 x \rfloor$, or, equivalently, $x = 2^k$.

Since all $(2^k, 2^k)$ pairs can be forced into $(1, 1)$ pairs, the first player wants to force situations when all pairs are $(1, 1)$ or empty.

J. Not Nim

Suppose that the first player will move in a pair that currently contains (x, x) stones. Forcing a second player's move here will result in $(x - 1, 0)$ instead of $(x, 0)$ (if the second player could skip). The move will be harmful if $\lfloor \log_2(x-1) \rfloor < \lfloor \log_2 x \rfloor$, or, equivalently, $x = 2^k$.

Since all $(2^k, 2^k)$ pairs can be forced into $(1, 1)$ pairs, the first player wants to force situations when all pairs are $(1, 1)$ or empty.

Forcing of a move happens when the first player empties the last heap in some pair. When this happens, we want all other pairs to contain equal heaps (otherwise, the second player can effectively skip).

J. Not Nim

Suppose that the first player will move in a pair that currently contains (x, x) stones. Forcing a second player's move here will result in $(x - 1, 0)$ instead of $(x, 0)$ (if the second player could skip). The move will be harmful if $\lfloor \log_2(x - 1) \rfloor < \lfloor \log_2 x \rfloor$, or, equivalently, $x = 2^k$.

Since all $(2^k, 2^k)$ pairs can be forced into $(1, 1)$ pairs, the first player wants to force situations when all pairs are $(1, 1)$ or empty.

Forcing of a move happens when the first player empties the last heap in some pair. When this happens, we want all other pairs to contain equal heaps (otherwise, the second player can effectively skip).

We want to make sizes of all pairs as close to $(1, 1)$ as possible without making unequal pairs. By making enough moves, a pair (x, x) can be forced to a pair $(2^k - 1, 2^k - 1)$, where k is the number of leading ones in binary representation of x. Moving further in this pair will result in unequal heaps. We suppose that at all times the heaps are reduced to this form.

J. Not Nim

Suppose that the second player moves in a $(2^k - 1, 2^k - 1)$ pair. If $k = 0$, then we empty the pair and the forcing continues. Otherwise, after dividing by two the heap turns into $(2^{k-1} - 1, 2^{k-1} - 1)$.

J. Not Nim

Suppose that the second player moves in a $(2^k - 1, 2^k - 1)$ pair. If $k = 0$, then we empty the pair and the forcing continues. Otherwise, after dividing by two the heap turns into $(2^{k-1} - 1, 2^{k-1} - 1)$.

On his move the first player has to choose $(2^k - 1, 2^k - 1)$ pair and empty it. At the end of this pair, the second player is then forced to make a move in another pair.

J. Not Nim

Suppose that the second player moves in a $(2^k - 1, 2^k - 1)$ pair. If $k = 0$, then we empty the pair and the forcing continues. Otherwise, after dividing by two the heap turns into $(2^{k-1} - 1, 2^{k-1} - 1)$.

On his move the first player has to choose $(2^k - 1, 2^k - 1)$ pair and empty it. At the end of this pair, the second player is then forced to make a move in another pair.

Observation

Both the first and the second player will choose a pair with maximal k.

J. Not Nim

Suppose that the second player moves in a $(2^k - 1, 2^k - 1)$ pair. If $k = 0$, then we empty the pair and the forcing continues. Otherwise, after dividing by two the heap turns into $(2^{k-1} - 1, 2^{k-1} - 1)$.

On his move the first player has to choose $(2^k - 1, 2^k - 1)$ pair and empty it. At the end of this pair, the second player is then forced to make a move in another pair.

Observation

Both the first and the second player will choose a pair with maximal k.

Indeed, the first player will want the second player to move in small pairs, so he'll eliminate the large ones.

J. Not Nim

Suppose that the second player moves in a $(2^k - 1, 2^k - 1)$ pair. If $k = 0$, then we empty the pair and the forcing continues. Otherwise, after dividing by two the heap turns into $(2^{k-1} - 1, 2^{k-1} - 1)$.

On his move the first player has to choose $(2^k - 1, 2^k - 1)$ pair and empty it. At the end of this pair, the second player is then forced to make a move in another pair.

Observation

Both the first and the second player will choose a pair with maximal k.

Indeed, the first player will want the second player to move in small pairs, so he'll eliminate the large ones.

Similarly, in the end the second player wants to have as few $(1, 1)$ pairs as possible, so he'll avoid making them from $(3, 3)$. Similarly, to avoid making $(3, 3)$ he'll avoid moving in $(7, 7)$, and so on.

J. Not Nim

Finally, we can model the game as follows. After reducing all heaps to $(2^k - 1, 2^k - 1)$ form we'll store the number of pairs with $k = 0, 1, \ldots, \log_2 A$ (where A is the maximal value of a heap size).

J. Not Nim

Finally, we can model the game as follows. After reducing all heaps to $(2^k - 1, 2^k - 1)$ form we'll store the number of pairs with $k = 0, 1, \ldots, \log_2 A$ (where A is the maximal value of a heap size).

The first player will choose k as large as possible and erase one $(2^k - 1)$-pair. Then the second player chooses the largest possible pair too.

J. Not Nim

Finally, we can model the game as follows. After reducing all heaps to $(2^k - 1, 2^k - 1)$ form we'll store the number of pairs with $k = 0, 1, \dots, \log_2 A$ (where A is the maximal value of a heap size).

The first player will choose k as large as possible and erase one $(2^k - 1)$-pair. Then the second player chooses the largest possible pair too.

If he has $k > 0$, then he turns this pair into $(2^{k-1} - 1)$ and the first player moves again.

J. Not Nim

Finally, we can model the game as follows. After reducing all heaps to $(2^k - 1, 2^k - 1)$ form we'll store the number of pairs with $k = 0, 1, \ldots, \log_2 A$ (where A is the maximal value of a heap size).

The first player will choose k as large as possible and erase one $(2^k - 1)$-pair. Then the second player chooses the largest possible pair too.

If he has $k > 0$, then he turns this pair into $(2^{k-1} - 1)$ and the first player moves again.

Otherwise, all the rest moves of the second player are forced.

J. Not Nim

Finally, we can model the game as follows. After reducing all heaps to $(2^k - 1, 2^k - 1)$ form we'll store the number of pairs with $k = 0, 1, \dots, \log_2 A$ (where A is the maximal value of a heap size).

The first player will choose k as large as possible and erase one $(2^k - 1)$-pair. Then the second player chooses the largest possible pair too.

If he has $k > 0$, then he turns this pair into $(2^{k-1} - 1)$ and the first player moves again.

Otherwise, all the rest moves of the second player are forced.

All preprocessing and the final game process can be implemented in $O(n \log A)$ time.

J. Not Nim

Finally, we can model the game as follows. After reducing all heaps to $(2^k - 1, 2^k - 1)$ form we'll store the number of pairs with $k = 0, 1, \dots, \log_2 A$ (where A is the maximal value of a heap size).

The first player will choose k as large as possible and erase one $(2^k - 1)$-pair. Then the second player chooses the largest possible pair too.

If he has $k > 0$, then he turns this pair into $(2^{k-1} - 1)$ and the first player moves again.

Otherwise, all the rest moves of the second player are forced.

All preprocessing and the final game process can be implemented in $O(n \log A)$ time.

If you have trouble understanding this solution, try to work out why the answer for $3, 3, 3$ input is 15 instead of 17.

K. Stutter

Find longest common subsequence of two sequences a and b that consists of pairs of equal numbers. You can perform $O(nm)$ operations, but can't store $\Omega(nm)$ memory.

K. Stutter

Find longest common subsequence of two sequences a and b that consists of pairs of equal numbers. You can perform $O(nm)$ operations, but can't store $\Omega(nm)$ memory.

Outline: optimize the memory with additional bookkeeping.

K. Stutter

The standard DP solution for LCS can be modified as follows. Let $dp_{i,j}$ be equal to the maximal length of LCS of first i and j elements of a and b respectively if we are forced to take both elements of each pair simultaneously.

K. Stutter

The standard DP solution for LCS can be modified as follows. Let $dp_{i,j}$ be equal to the maximal length of LCS of first i and j elements of a and b respectively if we are forced to take both elements of each pair simultaneously.

Let $prev(a, i, c)$ be the last occurence of c in a before position i. Then $dp_{i,j} = \max(dp_{i-1,j}, dp_{i,j-1})$ if $a_i \neq b_j$, and $\max(dp_{i-1,j}, dp_{i,j-1}, 2 + dp_{prev(a,i,c)-1,prev(b,j,c)-1})$ if $a_i = b_j = c$ (all $prev$'s have to be defined, of course).

K. Stutter

Let us compute $dp_{i,j}$ row by row. We can't store the whole matrix, so we'll just have two last rows.

K. Stutter

Let us compute $dp_{i,j}$ row by row. We can't store the whole matrix, so we'll just have two last rows.

To account for $dp_{prev(\ldots),prev(\ldots)}$ let us store

$$dp_j^{equal} = \max_{i | a_i = b_j} dp_{i-1,j-1},$$

where i ranges over all processed rows.

K. Stutter

Let us compute $dp_{i,j}$ row by row. We can't store the whole matrix, so we'll just have two last rows.

To account for $dp_{prev(\dots),prev(\dots)}$ let us store

$$dp_j^{equal} = \max_{i \mid a_i = b_j} dp_{i-1,j-1},$$

where i ranges over all processed rows.

Note that we can use $dp_{prev(b,j,c)}^{equal}$ in place of $dp_{prev(a,i,c)-1,prev(b,j,c)-1}$. This eliminates our need for $\Omega(nm)$ memory and requires only $O(n+m)$ memory.