Moving to Europe Analysis

September 24, 2019

Mikhail Tikhomirov, Pavel Kunyavsky, Maxim Akhmedov

Discover Singapore 2019

A. Beads

Given a sequence of n numbers, choose a positive integer k such that the partition of the first $n - (n \bmod k)$ numbers into contiguous parts of length k has the maximum number of distinct parts. A sequence and its reverse is considered equal.

A. Beads

Recall that we can compute a polynomial hash $h(t)$ of any substring t of a string in $O(1)$. To account for possible reverse, let us compute the hash of a substring t as an *unordered* pair $\{h(t), h(rev(t))\}$.

A. Beads

Recall that we can compute a polynomial hash $h(t)$ of any substring t of a string in $O(1)$. To account for possible reverse, let us compute the hash of a substring t as an *unordered* pair $\{h(t), h(rev(t))\}$.

For a particular k, there are $\lfloor n/k \rfloor$ relevant substrings. compute hashes of all of them and count distinct ones with sorting/set in $O(n/k \cdot \log n)$. To avoid birthday paradox may need to use several distinct hash modulos.

A. Beads

Recall that we can compute a polynomial hash $h(t)$ of any substring t of a string in $O(1)$. To account for possible reverse, let us compute the hash of a substring t as an *unordered* pair $\{h(t), h(rev(t))\}$.

For a particular k, there are $\lfloor n/k \rfloor$ relevant substrings. compute hashes of all of them and count distinct ones with sorting/set in $O(n/k \cdot \log n)$. To avoid birthday paradox may need to use several distinct hash modulos.

The total complexity is $O(n \log n \cdot \sum_{k=1}^{n} 1/k) = O(n \log^2 n)$.

B. Board Trick

We are given an undirected graph with weights for each direction of each edge. Find an Euler tour that minimizes the largest of used directions for edges.

B. Board Trick

Let us check if there is a tour with the largest weight at most x. The only condition we have to satisfy is that for each vertex its out-degree is equal to its in-degree. In particular, if any vertex has odd degree, clearly there is no answer.

B. Board Trick

Let us check if there is a tour with the largest weight at most x. The only condition we have to satisfy is that for each vertex its out-degree is equal to its in-degree. In particular, if any vertex has odd degree, clearly there is no answer.

For any undirected edge:

- it increases the in-degree of at most one of its endpoints;

B. Board Trick

Let us check if there is a tour with the largest weight at most x. The only condition we have to satisfy is that for each vertex its out-degree is equal to its in-degree. In particular, if any vertex has odd degree, clearly there is no answer.

For any undirected edge:

- it increases the in-degree of at most one of its endpoints;
- it can increase the in-degree of an endpoint if its direction towards it has weight at most x.

B. Board Trick

Construct a flow network as follows:

- the set of vertices is $\{s, t\} \cup E \cup V$, where s, t are the source and the sink, E and V are edges and vertices of the original graph;

B. Board Trick

Construct a flow network as follows:

- the set of vertices is $\{s, t\} \cup E \cup V$, where s, t are the source and the sink, E and V are edges and vertices of the original graph;
- for all $e \in E$, add (s, e) of weight 1;

B. Board Trick

Construct a flow network as follows:

- the set of vertices is $\{s, t\} \cup E \cup V$, where s, t are the source and the sink, E and V are edges and vertices of the original graph;
- for all $e \in E$, add (s, e) of weight 1;
- if $e = (v, u)$, then add (e, v) iff the direction of e towards v has weight at most x;

B. Board Trick

Construct a flow network as follows:

- the set of vertices is $\{s, t\} \cup E \cup V$, where s, t are the source and the sink, E and V are edges and vertices of the original graph;
- for all $e \in E$, add (s, e) of weight 1;
- if $e = (v, u)$, then add (e, v) iff the direction of e towards v has weight at most x;
- for all $v \in V$, add (v, t) of weight $deg(v)/2$.

B. Board Trick

Construct a flow network as follows:

- the set of vertices is $\{s, t\} \cup E \cup V$, where s, t are the source and the sink, E and V are edges and vertices of the original graph;
- for all $e \in E$, add (s, e) of weight 1;
- if $e = (v, u)$, then add (e, v) iff the direction of e towards v has weight at most x;
- for all $v \in V$, add (v, t) of weight $deg(v)/2$.

The flow from V to t corresponds to assignments of in-degree with all conditions in place. From above we have that the answer is at most x iff there is an $s - t$ flow of size m.

B. Board Trick

Construct a flow network as follows:

- the set of vertices is $\{s, t\} \cup E \cup V$, where s, t are the source and the sink, E and V are edges and vertices of the original graph;
- for all $e \in E$, add (s, e) of weight 1;
- if $e = (v, u)$, then add (e, v) iff the direction of e towards v has weight at most x;
- for all $v \in V$, add (v, t) of weight $deg(v)/2$.

The flow from V to t corresponds to assignments of in-degree with all conditions in place. From above we have that the answer is at most x iff there is an $s - t$ flow of size m.

Binary search on x. The graph is small enough so that advanced max-flow techniques are not needed.

C. Frog

There are n distinct points $p_1, \ldots, p_n$ in the real line. A *jump* for the point p_i consists of moving to the k-th closest point with respect to the i-th (breaking tie to prefer the leftmost point). Given a large m, for each i determine the location after m jumps starting from p_i.

C. Frog

Let S_i be the set of $k+1$ closest points to p_i (including itself, taking care of the tie), and let $L_i = \min S_i$, $R_i = \max S_i$. Observe that $L_i \leqslant L_j, R_i \leqslant R_j$ for any $i < j$.

C. Frog

Let S_i be the set of $k+1$ closest points to p_i (including itself, taking care of the tie), and let $L_i = \min S_i$, $R_i = \max S_i$. Observe that $L_i \leqslant L_j, R_i \leqslant R_j$ for any $i < j$.

Let us compute L_i, R_i from left to right. Clearly, $L_1 = 1, R_1 = k+1$. For any $i > 1$, set $L_i = L_{i-1}$, $R_i = R_{i-1}$ and increase both of them while $p_{R_i+1} - p_i < p_i - p_{L_i}$.

C. Frog

Let S_i be the set of $k+1$ closest points to p_i (including itself, taking care of the tie), and let $L_i = \min S_i$, $R_i = \max S_i$. Observe that $L_i \leqslant L_j, R_i \leqslant R_j$ for any $i < j$.

Let us compute L_i, R_i from left to right. Clearly, $L_1 = 1, R_1 = k+1$. For any $i > 1$, set $L_i = L_{i-1}$, $R_i = R_{i-1}$ and increase both of them while $p_{R_i+1} - p_i < p_i - p_{L_i}$.

A jump from p_i is either to p_{L_i} or p_{R_i}, whichever is closer.

C. Frog

Let S_i be the set of $k+1$ closest points to p_i (including itself, taking care of the tie), and let $L_i = \min S_i$, $R_i = \max S_i$. Observe that $L_i \leqslant L_j, R_i \leqslant R_j$ for any $i < j$.

Let us compute L_i, R_i from left to right. Clearly, $L_1 = 1, R_1 = k+1$. For any $i > 1$, set $L_i = L_{i-1}$, $R_i = R_{i-1}$ and increase both of them while $p_{R_i+1} - p_i < p_i - p_{L_i}$.

A jump from p_i is either to p_{L_i} or p_{R_i}, whichever is closer.

Finally, compute binary lifting $t_{i,j}$ = the position after 2^j jumps from i. Both precomputation and answering the queries can be done in $O(n \log m)$.

D. Godzilla

Given a directed graph, process offline queries:

- erase an edge;
- find the smallest size of a set of vertices S such that each vertex of the graph is reachable from a vertex of S.

D. Godzilla

Perform all queries backwards: add edges instead of erasing.

D. Godzilla

Perform all queries backwards: add edges instead of erasing.

Let us store the following data:

D. Godzilla

Perform all queries backwards: add edges instead of erasing.

Let us store the following data:

- some partition (DSU) of vertices $V_1, \ldots, V_k$ so that all vertices of each V_i are reachable from each other;

D. Godzilla

Perform all queries backwards: add edges instead of erasing.

Let us store the following data:

- some partition (DSU) of vertices $V_1, \dots, V_k$ so that all vertices of each V_i are reachable from each other;
- for each V_i store a pointer P_i to another component V_{P_i} such that V_i is reachable from V_{P_i} and V_{P_i} is a *source component* (i.e. there is not $V_k \neq V_{P_i}$ such that V_{P_i} is reachable from V_k);

D. Godzilla

Perform all queries backwards: add edges instead of erasing.

Let us store the following data:

- some partition (DSU) of vertices $V_1, \ldots, V_k$ so that all vertices of each V_i are reachable from each other;
- for each V_i store a pointer P_i to another component V_{P_i} such that V_i is reachable from V_{P_i} and V_{P_i} is a *source component* (i.e. there is not $V_k \neq V_{P_i}$ such that V_{P_i} is reachable from V_k);
- for each non-source component V_i store a list of incoming edges L_i that are not accounted for.

D. Godzilla

Perform all queries backwards: add edges instead of erasing.

Let us store the following data:

- some partition (DSU) of vertices $V_1, \ldots, V_k$ so that all vertices of each V_i are reachable from each other;
- for each V_i store a pointer P_i to another component V_{P_i} such that V_i is reachable from V_{P_i} and V_{P_i} is a *source component* (i.e. there is not $V_k \neq V_{P_i}$ such that V_{P_i} is reachable from V_k);
- for each non-source component V_i store a list of incoming edges L_i that are not accounted for.
- the number of source components.

D. Godzilla

Perform all queries backwards: add edges instead of erasing.

Let us store the following data:

- some partition (DSU) of vertices $V_1, \ldots, V_k$ so that all vertices of each V_i are reachable from each other;
- for each V_i store a pointer P_i to another component V_{P_i} such that V_i is reachable from V_{P_i} and V_{P_i} is a *source component* (i.e. there is not $V_k \neq V_{P_i}$ such that V_{P_i} is reachable from V_k);
- for each non-source component V_i store a list of incoming edges L_i that are not accounted for.
- the number of source components.

Additionally, we require that all source V_j are actual SCC's of the graph and for each of them L_j is empty.

D. Godzilla

How do we add an edge $a \to b$? Locate the components V_i, V_j containing a, b, and add a to L_j. If V_j is not a source component (that is, $P_j \neq j$), we don't have to do anything else.

D. Godzilla

How do we add an edge $a \to b$? Locate the components V_i, V_j containing a, b, and add a to L_j. If V_j is not a source component (that is, $P_j \neq j$), we don't have to do anything else.

Otherwise, start processing elements of L_j one by one (initially, a is the only element). Consider a few cases:

- if L_j is empty, then V_j is still a source and we can exit;

D. Godzilla

How do we add an edge $a \to b$? Locate the components V_i, V_j containing a, b, and add a to L_j. If V_j is not a source component (that is, $P_j \neq j$), we don't have to do anything else.

Otherwise, start processing elements of L_j one by one (initially, a is the only element). Consider a few cases:

- if L_j is empty, then V_j is still a source and we can exit;
- if $x \in V_j$, erase it and continue;

D. Godzilla

How do we add an edge $a \to b$? Locate the components V_i, V_j containing a, b, and add a to L_j. If V_j is not a source component (that is, $P_j \neq j$), we don't have to do anything else.

Otherwise, start processing elements of L_j one by one (initially, a is the only element). Consider a few cases:

- if L_j is empty, then V_j is still a source and we can exit;
- if $x \in V_j$, erase it and continue;
- if $x \in V_k \neq V_j$, consider P_k. If $P_k = j$, then merge V_j with V_k, unite L_j and L_k and continue;

D. Godzilla

How do we add an edge $a \to b$? Locate the components V_i, V_j containing a, b, and add a to L_j. If V_j is not a source component (that is, $P_j \neq j$), we don't have to do anything else.

Otherwise, start processing elements of L_j one by one (initially, a is the only element). Consider a few cases:

- if L_j is empty, then V_j is still a source and we can exit;
- if $x \in V_j$, erase it and continue;
- if $x \in V_k \neq V_j$, consider P_k. If $P_k = j$, then merge V_j with V_k, unite L_j and L_k and continue;
- if $P_k \neq j$, then decrease the number of source by 1, set $P_j = P_k$ and finish (V_j is not a source anymore).

D. Godzilla

How do we add an edge $a \to b$? Locate the components V_i, V_j containing a, b, and add a to L_j. If V_j is not a source component (that is, $P_j \neq j$), we don't have to do anything else.

Otherwise, start processing elements of L_j one by one (initially, a is the only element). Consider a few cases:

- if L_j is empty, then V_j is still a source and we can exit;
- if $x \in V_j$, erase it and continue;
- if $x \in V_k \neq V_j$, consider P_k. If $P_k = j$, then merge V_j with V_k, unite L_j and L_k and continue;
- if $P_k \neq j$, then decrease the number of source by 1, set $P_j = P_k$ and finish (V_j is not a source anymore).

All operations with L_j are amortized, thus the complexity is determined by $O(n + m)$ DSU operations, each of which are $O(\log(n + m))$ (or faster).

E. Intelligence test

Given a sequence a, answer queries "is b a subsequence of a?".

E. Intelligence test

Given a sequence a, answer queries "is b a subsequence of a?".

Greedy algorithm works: locate the earliest location i_1 of b_1, then the earliest location $i_2 > i_1$ of b_2, and so on. The answer is "no" if no suitable location exists at some point.

E. Intelligence test

Given a sequence a, answer queries "is b a subsequence of a?".

Greedy algorithm works: locate the earliest location i_1 of b_1, then the earliest location $i_2 > i_1$ of b_2, and so on. The answer is "no" if no suitable location exists at some point.

To find $i_1, i_2, \ldots$ use binary search in lists of occurences of each element. Total complexity is $O(S \log n)$, where S is the total length of queries.

F. Lamp

You are given two sets of windows on two parallel buildings. Consider a lamp at the ground on the first building, which lights up the second building. Windows perfectly reflect light. Which windows of the first buildings will get a ray of light at their interior?

F. Lamp

We will define a ray of light by the coordinates of the point (x, y) on the second building it landed to.

F. Lamp

We will define a ray of light by the coordinates of the point (x, y) on the second building it landed to.

A ray performs k reflections if $(x, y) \in C$, $(2x, 2y) \in B$, $(3x, 3y) \in C$, etc.

F. Lamp

We will define a ray of light by the coordinates of the point (x, y) on the second building it landed to.

A ray performs k reflections if $(x, y) \in C$, $(2x, 2y) \in B$, $(3x, 3y) \in C$, etc.

First, note that on each interesting trajectory there is at most 2000 reflections. Indeed, if $(x, y) \in B$ and (x, y) is a result of $2k$ reflections, then $(x', y') = (x/k, y/k) \in B$. Note that $(0, 0)$ does not belong to any rectangle in B, so $\max\{|x'|, y'\} \geqslant 1$. Thus, $\max\{|x|, y\} \geqslant k$. Considering the limits on x's and y's, it means that k is no more than 1000, so $2k$ is no more than 2000.

F. Lamp

Also, if ray (x, y) gets into (x', y') using k reflection where k is not a power of two, then another ray (x'', y'') gets there in k'' reflections, where k'' is a power of two.

F. Lamp

Also, if ray (x, y) gets into (x', y') using k reflection where k is not a power of two, then another ray (x'', y'') gets there in k'' reflections, where k'' is a power of two.

Indeed, let l be an odd divisor of k, then the ray $(x'', y'') = (lx, ly)$ gets into (x', y') in k/l reflections (as the corresponding progression of points will be a subprogession of the progression for (x, y)).

F. Lamp

Now we have to find out all rectangles, that may be attained in $k = 2^l$ reflections.

F. Lamp

Now we have to find out all rectangles, that may be attained in $k = 2^l$ reflections.

Consider all rectangles of sets $B/(2i)$ and $C/(2i+1)$ and find out all points covered by k rectangles using the scanline in $O(kn \log k)$

F. Lamp

Now we have to find out all rectangles, that may be attained in $k = 2^l$ reflections.

Consider all rectangles of sets $B/(2i)$ and $C/(2i+1)$ and find out all points covered by k rectangles using the scanline in $O(kn \log k)$

Total running time will be $O(nC \log(nC))$ where C is the maximum coordinate value.

G. Leonardo's Numbers

Compute $\sum_{i=0}^{n}(L_i)^k$, where $L_0 = L_1 = 2$, $L_i = L_{i-1} + L_{i-2} + 1$ for $i > 1$.

G. Leonardo's Numbers

Let v_i be a vector containing numbers $L_i^a L_{i-1}^b$ for all $0 \leqslant a + b \leqslant k$. After expanding trinomials in $L_i^a L_{i-1}^b = (L_{i-1} + L_{i-2} + 1)^a L_{i-1}^b$ one can find a matrix A such that $v_i = Av_{i-1}$.

G. Leonardo's Numbers

Let v_i be a vector containing numbers $L_i^a L_{i-1}^b$ for all $0 \leqslant a + b \leqslant k$. After expanding trinomials in $L_i^a L_{i-1}^b = (L_{i-1} + L_{i-2} + 1)^a L_{i-1}^b$ one can find a matrix A such that $v_i = A v_{i-1}$.

After adding an extra element accumulating $\sum L_i^k$, one can use fast matrix exponentiation to find the answer in $O(k^6 \log n)$ time.

H. Monotonicity

Given a sequence of numbers and an infinite periodic monotonicity pattern $c_1, c_2, \ldots$, find the largest subsequence that matches a prefix of the pattern.

H. Monotonicity

Let us use a greedy approach: for each position i store ans_i — the length of the largest subsequence ending at position i that matches a prefix of the pattern. To compute ans_i, we consider all $j < i$ and check if $a_j c_{ans_j} a_i$ holds, and in that case update ans_i with $ans_j + 1$.

H. Monotonicity

Let us use a greedy approach: for each position i store ans_i — the length of the largest subsequence ending at position i that matches a prefix of the pattern. To compute ans_i, we consider all $j < i$ and check if $a_j c_{ans_j} a_i$ holds, and in that case update ans_i with $ans_j + 1$.

This can be sped using segment trees.

H. Monotonicity

Let us use a greedy approach: for each position i store ans_i — the length of the largest subsequence ending at position i that matches a prefix of the pattern. To compute ans_i, we consider all $j < i$ and check if $a_j c_{ans_j} a_i$ holds, and in that case update ans_i with $ans_j + 1$.

This can be sped using segment trees.

But more importantly, why does it work??

H. Monotonicity

Assume the contrary, and consider the first position i such that $ans_i < L_i$, where L_i is the *actual* longest suitable subsequence length ending in i. Let us prove that the relaxation described above will put ans_i to at least L_i.

H. Monotonicity

Assume the contrary, and consider the first position i such that $ans_i < L_i$, where L_i is the *actual* longest suitable subsequence length ending in i. Let us prove that the relaxation described above will put ans_i to at least L_i.

Let a_j be the element preceding a_i in the optimal subsequence ending at i (thus, in particular, $a_j c_{L_i - 1} a_i$ holds). Note that $L_j \geqslant L_i - 1$, and if $L_j = L_i - 1$, then ans_i would be equal to L_i by updating directly from j. This means that $L_j \geqslant L_i$.

H. Monotonicity

Assume the contrary, and consider the first position i such that $ans_i < L_i$, where L_i is the *actual* longest suitable subsequence length ending in i. Let us prove that the relaxation described above will put ans_i to at least L_i.

Let a_j be the element preceding a_i in the optimal subsequence ending at i (thus, in particular, $a_j c_{L_i-1} a_i$ holds). Note that $L_j \geqslant L_i - 1$, and if $L_j = L_i - 1$, then ans_i would be equal to L_i by updating directly from j. This means that $L_j \geqslant L_i$.

Let $b_1, \ldots, b_{L_j} = a_j$ be the elements of an optimal subsequence ending at position j. Observe that if c_{L_i-1} is =, then $a_i = a_j$, and ans_i would receive a value at least $L_j \geqslant L_i$ by an update from b_{L_j-1} (since ans_j did).

Consider the sequence of operators $c_{L_i-1}, \ldots, c_{L_j-1}$. Suppose that at least one > and at least one < occurs in this sequence. Locate a closest pair c_x and c_y such that $c_x = <$, $c_y = >$, and only ='s happen in between. WLOG assume that $x < y$.

Consider the sequence of operators $c_{L_i-1}, \ldots, c_{L_j-1}$. Suppose that at least one > and at least one < occurs in this sequence. Locate a closest pair c_x and c_y such that $c_x = <$, $c_y = >$, and only ='s happen in between. WLOG assume that $x < y$.

We then have $b_x < b_{x+1} = \ldots = b_y$, thus either $b_x < a_i$ or $b_y > a_i$. In either case, an update from b_x or b_y would put ans_i to at least L_i. Thus, either there are no >'s or no <'s in the sequence (WLOG assume the former).

Consider the sequence of operators $c_{L_i-1}, \ldots, c_{L_j-1}$. Suppose that at least one > and at least one < occurs in this sequence. Locate a closest pair c_x and c_y such that $c_x = <$, $c_y = >$, and only ='s happen in between. WLOG assume that $x < y$.

We then have $b_x < b_{x+1} = \ldots = b_y$, thus either $b_x < a_i$ or $b_y > a_i$. In either case, an update from b_x or b_y would put ans_i to at least L_i. Thus, either there are no >'s or no <'s in the sequence (WLOG assume the former).

We then have a chain of comparisons $b_{L_i-1} c_{L_i-1} b_{L_i} c_{L_i} \ldots c_{L_j-1} a_j c_{L_i-1} a_i$. All operators here are < or =, and there is at least one < since c_{L_i-1} is not = (hence <), thus $b_{L_i-1} < a_i$. Thus updating from b_{L_i-1} puts ans_i to at least L_i, which completes the proof.

I. Sheep

Given a convex polygon with n vertices $P_1, \ldots, P_n$ in clockwise order, and k points inside of it, determine the number of triangulations of the polygon such that:

- no point lies on any diagonal;
- each triangle contains an even number of points.

I. Sheep

We'll start by counting the number $R_{a,b}$ of points to the right **or directly on** the directed diagonal $P_a P_b$ for all a, b. To do this fast enough, consider each given point q and update all $R_{a,b}$ accordingly.

I. Sheep

We'll start by counting the number $R_{a,b}$ of points to the right **or directly on** the directed diagonal P_aP_b for all a, b. To do this fast enough, consider each given point q and update all $R_{a,b}$ accordingly.

Given a, let's find b such that q is to the right or on P_aP_b and b is farthest away from a in cyclic order. We then need to add 1 to $R_{a,a+1}, \ldots, R_{a,b}$.

I. Sheep

We'll start by counting the number $R_{a,b}$ of points to the right **or directly on** the directed diagonal P_aP_b for all a, b. To do this fast enough, consider each given point q and update all $R_{a,b}$ accordingly.

Given a, let's find b such that q is to the right or on P_aP_b and b is farthest away from a in cyclic order. We then need to add 1 to $R_{a,a+1}, \ldots, R_{a,b}$.

We will instead store differences $R_{a,x+1} - R_{a,x}$, then only $O(1)$ differences need to be updated. The actual values can be restored in the end by prefix summing.

I. Sheep

We'll start by counting the number $R_{a,b}$ of points to the right **or directly on** the directed diagonal $P_a P_b$ for all a, b. To do this fast enough, consider each given point q and update all $R_{a,b}$ accordingly.

Given a, let's find b such that q is to the right or on $P_a P_b$ and b is farthest away from a in cyclic order. We then need to add 1 to $R_{a,a+1}, \ldots, R_{a,b}$.

We will instead store differences $R_{a,x+1} - R_{a,x}$, then only $O(1)$ differences need to be updated. The actual values can be restored in the end by prefix summing.

Observe that as a moves forward, b only moves forward as well and makes one full circle. This allows to process each point q in $O(n)$, and all given points in $O(nk)$ in total.

I. Sheep

Now let's get to counting triangulations. For all a, b compute $ways_{a,b}$ — the number of ways to triangulate the part of the polygon that lies to the right of P_aP_b. Put $ways_{a,a+1} = 1$ by definition.

I. Sheep

Now let's get to counting triangulations. For all a, b compute $ways_{a,b}$ — the number of ways to triangulate the part of the polygon that lies to the right of P_aP_b. Put $ways_{a,a+1} = 1$ by definition.

In any triangulation P_aP_b is a side of exactly one triangle, consider all options for the third point P_c in the cyclic segment $a, \ldots, b$. Then:

- None of the diagonals P_aP_b, P_aP_c, P_bP_c can contain any points. A diagonal P_aP_b contains a point iff $R_{a,b} + R_{b,a} > k$.

I. Sheep

Now let's get to counting triangulations. For all a, b compute $ways_{a,b}$ — the number of ways to triangulate the part of the polygon that lies to the right of P_aP_b. Put $ways_{a,a+1} = 1$ by definition.

In any triangulation P_aP_b is a side of exactly one triangle, consider all options for the third point P_c in the cyclic segment $a, \ldots, b$. Then:

- None of the diagonals P_aP_b, P_aP_c, P_bP_c can contain any points. A diagonal P_aP_b contains a point iff $R_{a,b} + R_{b,a} > k$.
- All parts have to contain an even number of points, that is, $R_{a,b}, R_{b,c}, R_{c,a}$ are even (since k is even, the triangle has an even number of points automatically).

I. Sheep

Now let's get to counting triangulations. For all a, b compute $ways_{a,b}$ — the number of ways to triangulate the part of the polygon that lies to the right of P_aP_b. Put $ways_{a,a+1} = 1$ by definition.

In any triangulation P_aP_b is a side of exactly one triangle, consider all options for the third point P_c in the cyclic segment $a, \ldots, b$. Then:

- None of the diagonals P_aP_b, P_aP_c, P_bP_c can contain any points. A diagonal P_aP_b contains a point iff $R_{a,b} + R_{b,a} > k$.
- All parts have to contain an even number of points, that is, $R_{a,b}, R_{b,c}, R_{c,a}$ are even (since k is even, the triangle has an even number of points automatically).

$ways_{a,b}$ is the sum of $ways_{a,c} \cdot ways_{c,b}$ over all suitable c. Compute $ways_{a,b}$ by increasing of the cyclic distance between a, b.

I. Sheep

Now let's get to counting triangulations. For all a, b compute $ways_{a,b}$ — the number of ways to triangulate the part of the polygon that lies to the right of P_aP_b. Put $ways_{a,a+1} = 1$ by definition.

In any triangulation P_aP_b is a side of exactly one triangle, consider all options for the third point P_c in the cyclic segment $a, \ldots, b$. Then:

- None of the diagonals P_aP_b, P_aP_c, P_bP_c can contain any points. A diagonal P_aP_b contains a point iff $R_{a,b} + R_{b,a} > k$.
- All parts have to contain an even number of points, that is, $R_{a,b}, R_{b,c}, R_{c,a}$ are even (since k is even, the triangle has an even number of points automatically).

$ways_{a,b}$ is the sum of $ways_{a,c} \cdot ways_{c,b}$ over all suitable c. Compute $ways_{a,b}$ by increasing of the cyclic distance between a, b. The total complexity is $O(nk + n^3)$.

J. Teleportation

You are given a graph, with distance from 1 to 2 at least 5. How many edges can be added, while keeping distance at least 5?

Let's run bfs from vertices 1 and 2, and denote sets

- A as vertices on distance 1 from vertex 1.
- B as vertices on distance 2 from vertex 1.
- C as vertices on distance 1 from vertex 2.
- D as vertices on distance 2 from vertex 2.
- E as all othere verices.

As initial distance big, this sets are not intersected.

J. Teleportation

One can show, that no edges between A and $C \cup D$ is allowed, no edges between C and $A \cup B$ is allowed, and no vertex can be connected to both A and C.

Also, if we add an edge to any vertex from verices 1 or 2, at least one other edge would be forbidden, so it's not usefull.

J. Teleportation

So, total answer is sum of

- $+\frac{n\cdot(n-1)}{2} - m$ — total number of edges
- $-|A||C|$ — edges between A and C are not allowed
- $-|A| - |C|$ — edges from 1 to C and from 2 to D are not allowed
- $-|A||D| - |B||C|$ — edges from A to D and from B to C are not allowed
- $-2 \cdot (|E| + |B| + |D|)$ — new edges from 1 and 2 are not allowed
- $+1$ — edge $1, 2$ was calculated twice on previouse step
- -min$(|A|, |C|) * |E|$ — as either A or C shouldn't be connected to each vertex