Grand Prix of Moscow Workshop Editorial

April 17 & 23, 2017

Mikhail Tikhomirov & Gleb Evstropov

Moscow Pre-Finals ACM ICPC Workshop, MIPT, 2017

A. Centroid Tree

Problem idea, developer: Konstantin Semenov
Editorial: Mikhail Tikhomirov

Distinguish a tree generated by a randomized Kruskal from a *centroid* tree of a random tree with very high precision.

A. Centroid Tree

Most statistics-based algorithms, like distinguishing by degrees, distances, and so on, will not have enough precision to be accepted.

A. Centroid Tree

Most statistics-based algorithms, like distinguishing by degrees, distances, and so on, will not have enough precision to be accepted.

The idea behind the solution is that a centroid tree has a very special structure that is very unlikely to appear in a random tree, provided the number of vertices is not too small.

A. Centroid Tree

Here is one successful approach for checking if a tree is a centroid tree:

- Note that the minimal-index centroid v of a tree T must stay a centroid of $C(T)$. Try all options for the original centroid v (there are most two).

A. Centroid Tree

Here is one successful approach for checking if a tree is a centroid tree:

- Note that the minimal-index centroid v of a tree T must stay a centroid of $C(T)$. Try all options for the original centroid v (there are most two).
- For each of the subtrees T_i its root u_i (adjacent to v) must be a centroid of T_i. Subsequently, all chlidren of u_i must be centroids of their corresponding trees, and so on. This results in an $O(n)$ DFS algorithm (assuming precomputed subtree sizes).

A. Centroid Tree

Here is one successful approach for checking if a tree is a centroid tree:

- Note that the minimal-index centroid v of a tree T must stay a centroid of $C(T)$. Try all options for the original centroid v (there are most two).
- For each of the subtrees T_i its root u_i (adjacent to v) must be a centroid of T_i. Subsequently, all chlidren of u_i must be centroids of their corresponding trees, and so on. This results in an $O(n)$ DFS algorithm (assuming precomputed subtree sizes).

We tried this solution on 10^6 random trees with 60 vertices each, and it never made a mistake!

B. Completely Multiplicative Function

Problem idea, developer, editorial: Mikhail Tikhomirov

Construct a completely multiplicative function (*c.m.f*) such that its first 10^6 prefix sums are within 20 by absolute value.

B. Completely Multiplicative Function

Note that each c.m.f. is defined unambigiously by its values on prime numbers (provided $f(1) \neq 0$). After deciding $f(p)$ for all prime p one can construct all the rest values of f in linear (or almost linear) time using the sieve.

B. Completely Multiplicative Function

Note that each c.m.f. is defined unambigiously by its values on prime numbers (provided $f(1) \neq 0$). After deciding $f(p)$ for all prime p one can construct all the rest values of f in linear (or almost linear) time using the sieve.

Choosing random values for $f(p)$ doesn't work (the balance goes out of control). However, it was possible to obtain a suitable set of values using sophisticated greedy approach or local optimization. Further, the source size limit allowed to cram the answer into the source code if it took too long to produce.

B. Completely Multiplicative Function

Note that each c.m.f. is defined unambigiously by its values on prime numbers (provided $f(1) \neq 0$). After deciding $f(p)$ for all prime p one can construct all the rest values of f in linear (or almost linear) time using the sieve.

Choosing random values for $f(p)$ doesn't work (the balance goes out of control). However, it was possible to obtain a suitable set of values using sophisticated greedy approach or local optimization. Further, the source size limit allowed to cram the answer into the source code if it took too long to produce.

Still, that wasn't an intended solution.

B. Completely Multiplicative Function

Let us put

$$f(p) = \begin{cases} -1, & \text{if } p = 3k + 2, \\ +1, & \text{if } p = 3 \text{ or } p = 3k + 1, \end{cases}$$

and ensure that the n-balance of f is $O(\log n)$.

B. Completely Multiplicative Function

Let us put

$$f(p) = \begin{cases} -1, & \text{if } p = 3k+2, \\ +1, & \text{if } p = 3 \text{ or } p = 3k+1, \end{cases}$$

and ensure that the n-balance of f is $O(\log n)$.

One can easily show that $f(3k+1) = 1$, $f(3k+2) = -1$, and $f(3k) = f(k)$.

B. Completely Multiplicative Function

Let us put

$$f(p) = \begin{cases} -1, & \text{if } p = 3k+2, \\ +1, & \text{if } p = 3 \text{ or } p = 3k+1, \end{cases}$$

and ensure that the n-balance of f is $O(\log n)$.

One can easily show that $f(3k+1) = 1$, $f(3k+2) = -1$, and $f(3k) = f(k)$.

Let us put $S(n) = f(1) + \ldots + f(n)$. We must have $S(3k+1) = 1 + S(k)$, and $S(3k) = S(3k+2) = S(k)$.

B. Completely Multiplicative Function

Let us put

$$f(p) = \begin{cases} -1, & \text{if } p = 3k+2, \\ +1, & \text{if } p = 3 \text{ or } p = 3k+1, \end{cases}$$

and ensure that the n-balance of f is $O(\log n)$.

One can easily show that $f(3k+1) = 1$, $f(3k+2) = -1$, and $f(3k) = f(k)$.

Let us put $S(n) = f(1) + \ldots + f(n)$. We must have $S(3k+1) = 1 + S(k)$, and $S(3k) = S(3k+2) = S(k)$.

It follows that $0 \leqslant S(n) \leqslant \log_3 n$ (in fact, $S(n)$ is exactly the number of 1's in the ternary expansion of n).

B. Completely Multiplicative Function

Let us put

$$f(p) = \begin{cases} -1, & \text{if } p = 3k+2, \\ +1, & \text{if } p = 3 \text{ or } p = 3k+1, \end{cases}$$

and ensure that the n-balance of f is $O(\log n)$.

One can easily show that $f(3k+1) = 1$, $f(3k+2) = -1$, and $f(3k) = f(k)$.

Let us put $S(n) = f(1) + \ldots + f(n)$. We must have $S(3k+1) = 1 + S(k)$, and $S(3k) = S(3k+2) = S(k)$.

It follows that $0 \leqslant S(n) \leqslant \log_3 n$ (in fact, $S(n)$ is exactly the number of 1's in the ternary expansion of n).

One can obtain a similar construction with assigning $f(p) = -1$ iff p mod m is a square non-residue modulo (not too large) m.

C. Even and Odd

Problem idea: Mikhail Tikhomirov
Developer: Ivan Smirnov
Editorial: Mikhail Tikhomirov

Count the number of pairs of vertices v, u in a connected undirected graph such that each $v - u$ path has even/odd number of edges.

C. Even and Odd

First of all, in a bipartite graph all paths connecting a pair of vertices have the same parity (even if the vertices are in the same half, and odd otherwise).

C. Even and Odd

First of all, in a bipartite graph all paths connecting a pair of vertices have the same parity (even if the vertices are in the same half, and odd otherwise).

However, in general case the answer for a particular pair of vertices may depend on which parts of the graph the path must be passing through.

C. Even and Odd

First of all, in a bipartite graph all paths connecting a pair of vertices have the same parity (even if the vertices are in the same half, and odd otherwise).

However, in general case the answer for a particular pair of vertices may depend on which parts of the graph the path must be passing through.

Recall that a *biconnected component* (or a *block*) of a graph is a maximal subset of equivalent edges under equivalency relation "edges e_1 and e_2 share a simple cycle".

C. Even and Odd

First of all, in a bipartite graph all paths connecting a pair of vertices have the same parity (even if the vertices are in the same half, and odd otherwise).

However, in general case the answer for a particular pair of vertices may depend on which parts of the graph the path must be passing through.

Recall that a *biconnected component* (or a *block*) of a graph is a maximal subset of equivalent edges under equivalency relation "edges e_1 and e_2 share a simple cycle".

Articulation points separate the graph into a *block-cut tree*. Using the block-cut tree, one can determine for each pair of vertices v, u which blocks any $v - u$ path must pass through.

C. Even and Odd

If all blocks between v and u are bipartite, it follows that $v - u$ paths have fixed parity.

C. Even and Odd

If all blocks between v and u are bipartite, it follows that $v - u$ paths have fixed parity.

Otherwise, suppose that at least one non-bipartite block occurs between v and u. First, we assert that

Claim

If v and u are distinct vertices of a non-bipartite block, there is an even-length $v - u$ path and an odd-length $v - u$ path.

The claim above can be proven with case analysis.

C. Even and Odd

If all blocks between v and u are bipartite, it follows that $v - u$ paths have fixed parity.

Otherwise, suppose that at least one non-bipartite block occurs between v and u. First, we assert that

Claim

If v and u are distinct vertices of a non-bipartite block, there is an even-length $v - u$ path and an odd-length $v - u$ path.

The claim above can be proven with case analysis.

Now consider any path passing through a non-bipartite block B. Due to the claim above, we may exchange the subpath inside B to a subpath of different parity. It follows that no fixed-parity pairs are on different sides of B, or partially inside B.

C. Even and Odd

Subsequently, for the purpose of the problem we may delete all edges of a non-bipartite block (we don't include thusly disconnected vertices in the answer).

C. Even and Odd

Subsequently, for the purpose of the problem we may delete all edges of a non-bipartite block (we don't include thusly disconnected vertices in the answer).

After we have erased all non-bipartite blocks, the graph falls apart into several bipartite connected components; the answer for this case is found trivially (see above).

C. Even and Odd

Subsequently, for the purpose of the problem we may delete all edges of a non-bipartite block (we don't include thusly disconnected vertices in the answer).

After we have erased all non-bipartite blocks, the graph falls apart into several bipartite connected components; the answer for this case is found trivially (see above).

To sum up: we produce the block decomposition of the graph, erase all edges of non-bipartite blocks, and find the answer for the remaining bipartite graph.

C. Even and Odd

Subsequently, for the purpose of the problem we may delete all edges of a non-bipartite block (we don't include thusly disconnected vertices in the answer).

After we have erased all non-bipartite blocks, the graph falls apart into several bipartite connected components; the answer for this case is found trivially (see above).

To sum up: we produce the block decomposition of the graph, erase all edges of non-bipartite blocks, and find the answer for the remaining bipartite graph.

Note that we don't even actually need the explicit block-cut tree. The blocks can be found with simple DFS (modification of articulation points finding algorithm with a stack). The complexity is $O(n + m)$.

D. Great Again

Problem idea: Mikhail Tikhomirov
Developer: Alexander Ostanin
Editorial: Mikhail Tikhomirov

Divide the segment region with elements taking values in $\{-1, 0, 1\}$ so that to maximize (the number of positive-sum segments) - (the number of negative-sum segments).

D. Great Again

There is a straightforward, but slow DP solution. Let ans_k be the answer for the first k elements, and p_k be the sum of the first k elements. Then we must have

$$dp_k = \max_{j=k-r}^{k-l} dp_j + \operatorname{sgn}(p_k - p_j)$$

(we assume that $dp_0 = 0$ and $dp_k = -\infty$ for negative k).

D. Great Again

There is a straightforward, but slow DP solution. Let ans_k be the answer for the first k elements, and p_k be the sum of the first k elements. Then we must have

$$dp_k = \max_{j=k-r}^{k-l} dp_j + \operatorname{sgn}(p_k - p_j)$$

(we assume that $dp_0 = 0$ and $dp_k = -\infty$ for negative k).

To avoid slow linear computation of dp_k we use some data structures. For particular k and b let us put

$$val_{k,b} = \max_{k-r \leqslant j \leqslant k-l, p_j = b} dp_j$$

(note that the maximum on the right-hand side may be taken over an empty set, by a common convention we then put $val_{k,b} = -\infty$).

D. Great Again

We now have

$$dp_k = \max\left(\max_{b<p_k}(val_{k,b}-1), \max_{b=p_k} val_{k,b}, \max_{b>p_k}(val_{k,b}+1)\right).$$

D. Great Again

We now have

$$dp_k = \max\left(\max_{b<p_k}(val_{k,b}-1), \max_{b=p_k} val_{k,b}, \max_{b>p_k}(val_{k,b}+1)\right).$$

We will store $val_{k,b}$ in an RMQ structure with b as indices. If the values of $val_{k,b}$ are up to date, computing dp_k is just a few range minimum queries.

D. Great Again

We now have

$$dp_k = \max\left(\max_{b<p_k}(val_{k,b}-1), \max_{b=p_k} val_{k,b}, \max_{b>p_k}(val_{k,b}+1)\right).$$

We will store $val_{k,b}$ in an RMQ structure with b as indices. If the values of $val_{k,b}$ are up to date, computing dp_k is just a few range minimum queries.

How do we update $val_{k,b}$ when k changes? Note that at most element enters the range $[k-r, k-l]$, and at most one element leaves. Let us store a queue of relevant positions for each value of b (note that b can be negative), now $val_{k,b}$ is equal to maximum in the b-th queue.

D. Great Again

Maintaining maximum in a queue can be done, for example, with a deque of elements that are greater than everyone after them. Clearly, the maximal element is at the front of the deque.

D. Great Again

Maintaining maximum in a queue can be done, for example, with a deque of elements that are greater than everyone after them. Clearly, the maximal element is at the front of the deque.

To push x, popback everyone at most x from the deque, and push x to the back. To pop an element, check if it is at the front of the deque and possibly popfront it. This approach has amortized linear complexity.

D. Great Again

Maintaining maximum in a queue can be done, for example, with a deque of elements that are greater than everyone after them. Clearly, the maximal element is at the front of the deque.

To push x, popback everyone at most x from the deque, and push x to the back. To pop an element, check if it is at the front of the deque and possibly popfront it. This approach has amortized linear complexity.

RMQ is the hardest part, thus the complexity is thus $O(n \log n)$.

E. Jumping is Fun

Problem idea: Gleb Evstropov
Developer: Ivan Smirnov
Editorial: Gleb Evstropov

Given a sequence $a_1, a_2, \ldots, a_n$ we define a directed graph $G = (V, E)$, where $|V| = n$ and there is an arc from i to j iff $|j - i| \leqslant a_i$. Find a pair (i, j), such that $\min(\rho(i, j), \rho(j, i))$ is maximum possible. Here, $\rho(i, j)$ denotes the number of arcs on the shortest path from i to j.

E. Jumping is Fun

First we would like to find a way to process the distance query faster than in linear time.

E. Jumping is Fun

First we would like to find a way to process the distance query faster than in linear time.

Observation: the set of nodes that can be reached from node i in k steps is a consecutive segment. Denote as $l_k(i)$ its leftmost node and as $r_k(i)$ its rightmost node. Knowing l_k and r_k we can compute

$$l_{k+1}(i) = \max(1, \min_{l_k(i) \leqslant j \leqslant r_k(i)} j - a_j)$$

and

$$r_{k+1}(i) = \min(n, \max_{l_k(i) \leqslant j \leqslant r_k(i)} j + a_j)$$

E. Jumping is Fun

To compute and keep all the above functions $l_k(i)$ we need up to $O(n^2)$ time and memory. Instead, we can use the idea similar to sparse tables and binary jumps for LCA, and compute $l_k(i)$ and $r_k(i)$ for all k that are powers of 2. Indeed,

$$l_{2k}(i) = \max(1, \min_{l_k(i) \leqslant j \leqslant r_k(i)} l_k(j)$$

and

$$r_{2k}(i) = \min(n, \max_{l_k(i) \leqslant j \leqslant r_k(i)} r_k(j)$$

E. Jumping is Fun

To compute and keep all the above functions $l_k(i)$ we need up to $O(n^2)$ time and memory. Instead, we can use the idea similar to sparse tables and binary jumps for LCA, and compute $l_k(i)$ and $r_k(i)$ for all k that are powers of 2. Indeed,

$$l_{2k}(i) = \max(1, \min_{l_k(i) \leqslant j \leqslant r_k(i)} l_k(j)$$

and

$$r_{2k}(i) = \min(n, \max_{l_k(i) \leqslant j \leqslant r_k(i)} r_k(j)$$

Using the segment tree to query minimum we can run the above computations in $O(n \log^2 n)$ time. Now, to process the query of minimum distance from i to j (consider $i < j$) we have to find minimum k, such that $r_k(i) \geqslant j$. This can be done in $O(\log^2)$ similar to binary jumps algorithm for LCA and LA problems.

E. Jumping is Fun

Now we would like to solve the problem for a fixed value of distance k. Does there exists a pair (i, j), such that $\min(\rho(i, j), \rho(j, i)) \geqslant k$?

E. Jumping is Fun

Now we would like to solve the problem for a fixed value of distance k. Does there exists a pair (i, j), such that $\min(\rho(i, j), \rho(j, i)) \geqslant k$?

Consider triples $(l_k(i), i, r_k(i))$. We are looking for a pair of triples $(l_k(i), i, r_k(i))$ and $(l_k(j), j, r_k(j))$ such that $i < l_k(j)$ and $r_k(i) < j$. For each j we can query minimum value of $r_k(i)$ on the prefix from 1 to $l_k(j) - 1$. This can be done in linear time after all triples are built.

E. Jumping is Fun

Now we would like to solve the problem for a fixed value of distance k. Does there exists a pair (i, j), such that $\min(\rho(i, j), \rho(j, i)) \geqslant k$?

Consider triples $(l_k(i), i, r_k(i))$. We are looking for a pair of triples $(l_k(i), i, r_k(i))$ and $(l_k(j), j, r_k(j))$ such that $i < l_k(j)$ and $r_k(i) < j$. For each j we can query minimum value of $r_k(i)$ on the prefix from 1 to $l_k(j) - 1$. This can be done in linear time after all triples are built.

The above solution requires $\log n$ iterations of binary search and $O(n \log^2 n)$ time to compute triples, thus $O(n \log^3 n)$ time in total. Note, that we can avoid recomputing triples from scratch every time using the fact that we actually go from largest powers of two to lowest (same as in LCA or LA). Thus, the running time will be reduced to $O(n \log^2 n)$.

F. Online LCS

Problem idea: Gleb Evstropov
Developer: Konstantin Semyonov
Editorial: Gleb Evstropov

Given two initially empty strings s and t you should process online q queries. Each query is to add '$0\widehat{}$' or '$1\widehat{}$' at the end of one of these strings and return the current length of the longest common substring of s and t.

F. Online LCS

Note, that if the current length of the longest common substring is k and we add a new character to the string s, the answer can increase only by 1, i.e. become equal to $k + 1$ and this might happen only if the suffix of string s of length $k + 1$ will be a substring of t.

F. Online LCS

Note, that if the current length of the longest common substring is k and we add a new character to the string s, the answer can increase only by 1, i.e. become equal to $k + 1$ and this might happen only if the suffix of string s of length $k + 1$ will be a substring of t.

First we would like to introduce the solution to the following problem: given a static string t and some p that is know to be its substing, be able to add character c at the beginning of p or at the end of p if $c + p$ or $p + c$ (respectively) is still a substring of t. Also, we would like to be able to remove first character of p.

F. Online LCS

Note, that if the current length of the longest common substring is k and we add a new character to the string s, the answer can increase only by 1, i.e. become equal to $k + 1$ and this might happen only if the suffix of string s of length $k + 1$ will be a substring of t.

First we would like to introduce the solution to the following problem: given a static string t and some p that is know to be its substing, be able to add character c at the beginning of p or at the end of p if $c + p$ or $p + c$ (respectively) is still a substring of t. Also, we would like to be able to remove first character of p.

Build a suffix automaton of string t. For string p we only need to keep the pointer to the current state v of automaton corresponding to p and $m = |p|$.

F. Online LCS

If the new character c is added at the end of p we should check whether there exists the corresponding arc from state v. If the character c is added at the beginning of p, we should first check whether p is the longest string corresponding to state v, i.e. whether $m = maxlen(v)$.

F. Online LCS

If the new character c is added at the end of p we should check whether there exists the corresponding arc from state v. If the character c is added at the beginning of p, we should first check whether p is the longest string corresponding to state v, i.e. whether $m = maxlen(v)$.

If $m < maxlen(v)$ simply check that c is equal to character at corresponding position, otherwise we should find the suffix link going to v using character c. To check this we need to maintain two graphs, one of arcs of suffix automaton and the other of reversed suffix links of automaton.

F. Online LCS

If the new character c is added at the end of p we should check whether there exists the corresponding arc from state v. If the character c is added at the beginning of p, we should first check whether p is the longest string corresponding to state v, i.e. whether $m = maxlen(v)$.

If $m < maxlen(v)$ simply check that c is equal to character at corresponding position, otherwise we should find the suffix link going to v using character c. To check this we need to maintain two graphs, one of arcs of suffix automaton and the other of reversed suffix links of automaton.

Finally, to remove a single character from the beginning of p we do nothing or rollback using suffix link. Note that all operations are performed in $O(1)$ time.

F. Online LCS

Moreover, we can add a new operation to this subproblem: add a new character at the end of the string t, as the algorithm of suffix automaton in online itself. The only thing that might change about v is that the state v can be split in two (clone operation in automaton), then we should pick the right one assuming length of p.

F. Online LCS

Moreover, we can add a new operation to this subproblem: add a new character at the end of the string t, as the algorithm of suffix automaton in online itself. The only thing that might change about v is that the state v can be split in two (clone operation in automaton), then we should pick the right one assuming length of p.

Now, the final step is to be able to adopt the above algorithm to check the suffix of string s of length $k+1$ to be a substring of t. We would keep as a string p from the previous slide any substring $s(i,j)$, such that $|s|-k \leqslant i \leqslant j \leqslant |s|$, $s(i,j)$ is a substring of t, $i = |s|-k$ or $s(i-1,j)$ is not a substring of t, and $j = |s|$ or $s(i,j+1)$ is not a substring of t. Informal, $s(i,j)$ is maximal in some sense, i.e. it cannot be greedily expanded.

F. Online LCS

Obviously, if $s(|s| - k, |s|)$ is a substring of t, any its substring is a substring of t, thus it would be the only valid maximal substring and we will detect the case of answer increment.

F. Online LCS

Obviously, if $s(|s| - k, |s|)$ is a substring of t, any its substring is a substring of t, thus it would be the only valid maximal substring and we will detect the case of answer increment.

From the other hand, the length of a valid substring $s(i, j)$ never decreases by more than 1 in one operation. This only happens if $i = |s| - k$ and a new character is appended to s but the answer doesn't change. Thus, the amorhtized running time of one operation is $O(1)$ and the total running time is $O(q)$.

G. Brawling

Problem idea: Mikhail Tikhomirov
Developer: Nikita Uvarov
Editorial: Mikhail Tikhomirov

There are n people in a row, two people facing each other may fight which results in pushing one of them out of the row. What is the smallest number of people that can be left after a sequence of fights?

G. Brawling

Obviously, if several leftmost people are facing left, they cannot take part in fights, hence they will stay there forever; same for the rightmost people facing right. If there are no more people other than mentioned above, the answer is n.

G. Brawling

Obviously, if several leftmost people are facing left, they cannot take part in fights, hence they will stay there forever; same for the rightmost people facing right. If there are no more people other than mentioned above, the answer is n.

Otherwise, there is a group of people in the middle, so that the situation looks like:

LLL**R**...**L**RRR

G. Brawling

Obviously, if several leftmost people are facing left, they cannot take part in fights, hence they will stay there forever; same for the rightmost people facing right. If there are no more people other than mentioned above, the answer is n.

Otherwise, there is a group of people in the middle, so that the situation looks like:

LLL**R**...**L**RRR

It is easy to see that the middle group can not be cleared, but is possible to leave only one person there: kill all R's except for the leftmost one, then kill all L's with the leftmost one. The answer is then the total size of the extreme groups plus one.

H. I Spy

Author, developer: Alexander Golovanov
Editorial: Mikhail Tikhomirov

This is an interactive problem.

n points $p_1, \ldots, p_n$ with integer coordinates are placed in the plane. We can ask queries "how many points are within distance $\sqrt{R}$ of the point (x, y)?" with integer x, y, R. Guess all n points with a sufficiently small number of queries.

H. I Spy

Let $q = (x, y)$ be any point. Let us determine the unordered (multi)set of (squared) distances $||q - p_1||^2, \ldots, ||q - p_n||^2$.

H. I Spy

Let $q = (x, y)$ be any point. Let us determine the unordered (multi)set of (squared) distances $||q - p_1||^2, \ldots, ||q - p_n||^2$.

Let us do a binary search to find minimal R so that k of the given initial points are within $\sqrt{R}$. Now do this for all $k = 1, \ldots, n$.

H. I Spy

Let $q = (x, y)$ be any point. Let us determine the unordered (multi)set of (squared) distances $||q - p_1||^2, \ldots, ||q - p_n||^2$.

Let us do a binary search to find minimal R so that k of the given initial points are within $\sqrt{R}$. Now do this for all $k = 1, \ldots, n$.

How many queries would we need? Each of n times we do an integer binary search within a range $[0, 4 \cdot 10^{12}]$ (the maximal value of square of a distance).

H. I Spy

Let $q = (x, y)$ be any point. Let us determine the unordered (multi)set of (squared) distances $||q - p_1||^2, \ldots, ||q - p_n||^2$.

Let us do a binary search to find minimal R so that k of the given initial points are within $\sqrt{R}$. Now do this for all $k = 1, \ldots, n$.

How many queries would we need? Each of n times we do an integer binary search within a range $[0, 4 \cdot 10^{12}]$ (the maximal value of square of a distance).

Hence, the number of operations is roughly Kn, where $K = \log_2(4 \cdot 10^{12}) < 42$.

H. I Spy

Intended solution (unoptimized): choose three random points q_1, q_2, q_3 and find unordered distances from each of them to each initial point.

H. I Spy

Intended solution (unoptimized): choose three random points q_1, q_2, q_3 and find unordered distances from each of them to each initial point.

There are now n circles centered at each of q_1, q_2, q_3. Any point p_i must lie on a triple intersection of some of these circles. We can explicitly find each intersection r of circles centered at q_1 and q_2, and check that:

H. I Spy

Intended solution (unoptimized): choose three random points q_1, q_2, q_3 and find unordered distances from each of them to each initial point.

There are now n circles centered at each of q_1, q_2, q_3. Any point p_i must lie on a triple intersection of some of these circles. We can explicitly find each intersection r of circles centered at q_1 and q_2, and check that:

- r lies on a circle centered at q_3,

H. I Spy

Intended solution (unoptimized): choose three random points q_1, q_2, q_3 and find unordered distances from each of them to each initial point.

There are now n circles centered at each of q_1, q_2, q_3. Any point p_i must lie on a triple intersection of some of these circles. We can explicitly find each intersection r of circles centered at q_1 and q_2, and check that:

- r lies on a circle centered at q_3,
- r is (sufficiently close to) an integer point.

H. I Spy

Intended solution (unoptimized): choose three random points q_1, q_2, q_3 and find unordered distances from each of them to each initial point.

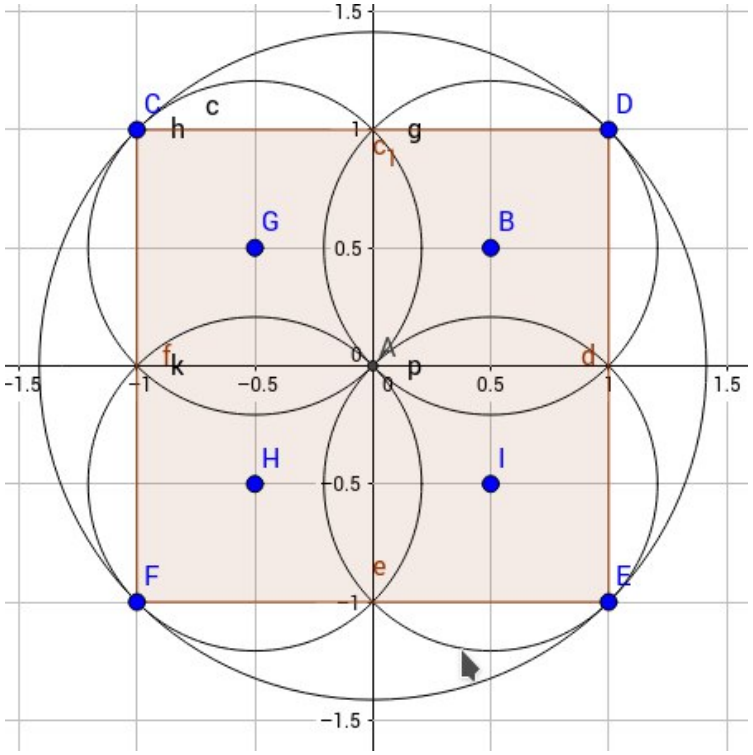
There are now n circles centered at each of q_1, q_2, q_3. Any point p_i must lie on a triple intersection of some of these circles. We can explicitly find each intersection r of circles centered at q_1 and q_2, and check that:

- r lies on a circle centered at q_3,
- r is (sufficiently close to) an integer point.

Finally, one may verify that all resulting points are among initial points by making a $R = 0$ query (this is hardly needed since coincidences are highly unlikely if q_1, q_2, q_3 are random). The number of queries is bounded by $3Kn + n < 13500$.

H. I Spy

Better solution (by Nikita Uvarov): perform a quad-tree like descent, with circles covering corresponding squares. Stop descending once the current circle contains no points.



H. I Spy

Note that at each step of the descent the radius of a circle is halved. It is convenient to start with $r = 2^m > \sqrt{2} \cdot 10^6$ so that the initial circle covers all points. Once the radius reaches $r = 2$, try all points inside the circle explicitly.

H. I Spy

Note that at each step of the descent the radius of a circle is halved. It is convenient to start with $r = 2^m > \sqrt{2} \cdot 10^6$ so that the initial circle covers all points. Once the radius reaches $r = 2$, try all points inside the circle explicitly.

How to bound the number of queries? Let us notice that it does not exceed the number of incidence pairs (p_i, C_j), where p_i is an initial point, C_j is one of the circles occuring in the quad-tree covering p_i.

H. I Spy

Note that at each step of the descent the radius of a circle is halved. It is convenient to start with $r = 2^m > \sqrt{2} \cdot 10^6$ so that the initial circle covers all points. Once the radius reaches $r = 2$, try all points inside the circle explicitly.

How to bound the number of queries? Let us notice that it does not exceed the number of incidence pairs (p_i, C_j), where p_i is an initial point, C_j is one of the circles occuring in the quad-tree covering p_i.

But the number of such pairs is at most $4mn$. Indeed, each point is covered by at most four circles on each layer of recursion.

H. I Spy

Note that at each step of the descent the radius of a circle is halved. It is convenient to start with $r = 2^m > \sqrt{2} \cdot 10^6$ so that the initial circle covers all points. Once the radius reaches $r = 2$, try all points inside the circle explicitly.

How to bound the number of queries? Let us notice that it does not exceed the number of incidence pairs (p_i, C_j), where p_i is an initial point, C_j is one of the circles occuring in the quad-tree covering p_i.

But the number of such pairs is at most $4mn$. Indeed, each point is covered by at most four circles on each layer of recursion.

Substituting $m \sim 20$, we obtain a solution that makes roughly 8000 queries (not counting $9n \leqslant 900$ queries at the bottom layer).

H. I Spy

Yet another solution (by contestants): find the set of distances from a random point q, then for each squared distance r_i try *all* integer points at distance $\sqrt{r_i}$ from q.

H. I Spy

Yet another solution (by contestants): find the set of distances from a random point q, then for each squared distance r_i try *all* integer points at distance $\sqrt{r_i}$ from q.

All integer points on a circle of radius $\sqrt{r_i}$ can be found in $O(\sqrt{r_i}) = O(M)$, where $M = 10^6$ is the maximum absolute value of coordinate. We need to perform nM operations just to find all candidates.

H. I Spy

Yet another solution (by contestants): find the set of distances from a random point q, then for each squared distance r_i try *all* integer points at distance $\sqrt{r_i}$ from q.

All integer points on a circle of radius $\sqrt{r_i}$ can be found in $O(\sqrt{r_i}) = O(M)$, where $M = 10^6$ is the maximum absolute value of coordinate. We need to perform nM operations just to find all candidates.

Can we estimate the number of queries this solution makes?

H. I Spy

Yet another solution (by contestants): find the set of distances from a random point q, then for each squared distance r_i try *all* integer points at distance $\sqrt{r_i}$ from q.

All integer points on a circle of radius $\sqrt{r_i}$ can be found in $O(\sqrt{r_i}) = O(M)$, where $M = 10^6$ is the maximum absolute value of coordinate. We need to perform nM operations just to find all candidates.

Can we estimate the number of queries this solution makes?

Not really, but it is small and fits within the limit nicely.

I. Rage Minimum Query

Problem idea: Gleb Evstropov
Developer: Ivan Smirnov
Editorial: Gleb Evstropov

You are given an array of size n. Initially, all elements are set to ∞. Then m consecutive queries take place. Each query is of form "set element i to be equal to x". After each query we have to report the current minimum value in the array. This problem is classic and an $O(m \log n)$ solution is known. However, in this problem we are guaranteed that queries are random and equiprobable (all valid i and all valid x) and we are asked to find an $O(n + m)$ solution.

I. Rage Minimum Query

Denote the current minimim as v and its position as p. Note, that if the element being changed is not p the new minimum is value $\min(v, x)$. Thus, if $i \neq p$ we recompute minimum in $O(1)$ time.

I. Rage Minimum Query

Denote the current minimim as v and its position as p. Note, that if the element being changed is not p the new minimum is value $\min(v, x)$. Thus, if $i \neq p$ we recompute minimum in $O(1)$ time.

Assuming i is selected equiprobably among $1, 2, \ldots, n$, the probability that $i = p$ is $\frac{1}{n}$. If that happens, we can simply select the minimum in linear time. Thus, the expected running time of a single operation is $\frac{n-1}{n} \cdot O(1) + \frac{1}{n} \cdot O(n) = O(1)$, and the overall expected running time is $O(m)$. Note, that we even didn't use that value of x is also selected at random.

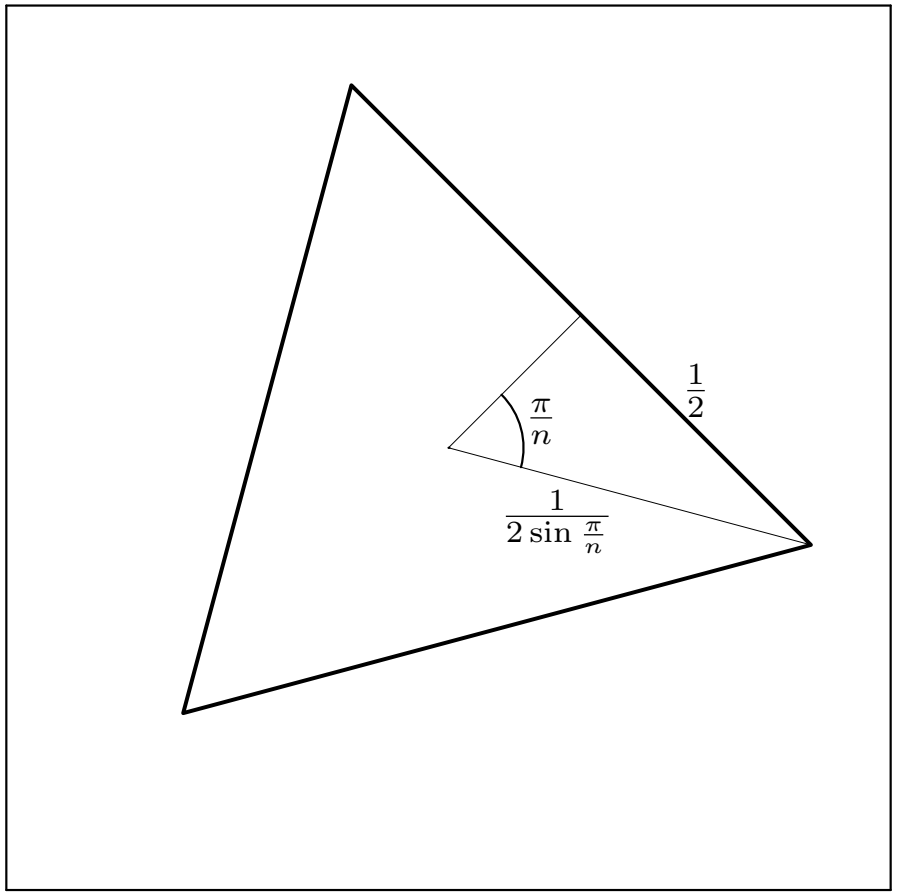
J. Regular Cake

Problem idea: Mikhail Tikhomirov
Developer: Alexander Golovanov
Editorial: Mikhail Tikhomirov

There is a regular n-gon with side length 1. Find the smallest side length of a regular m-gon with coinciding center and containing the n-gone.

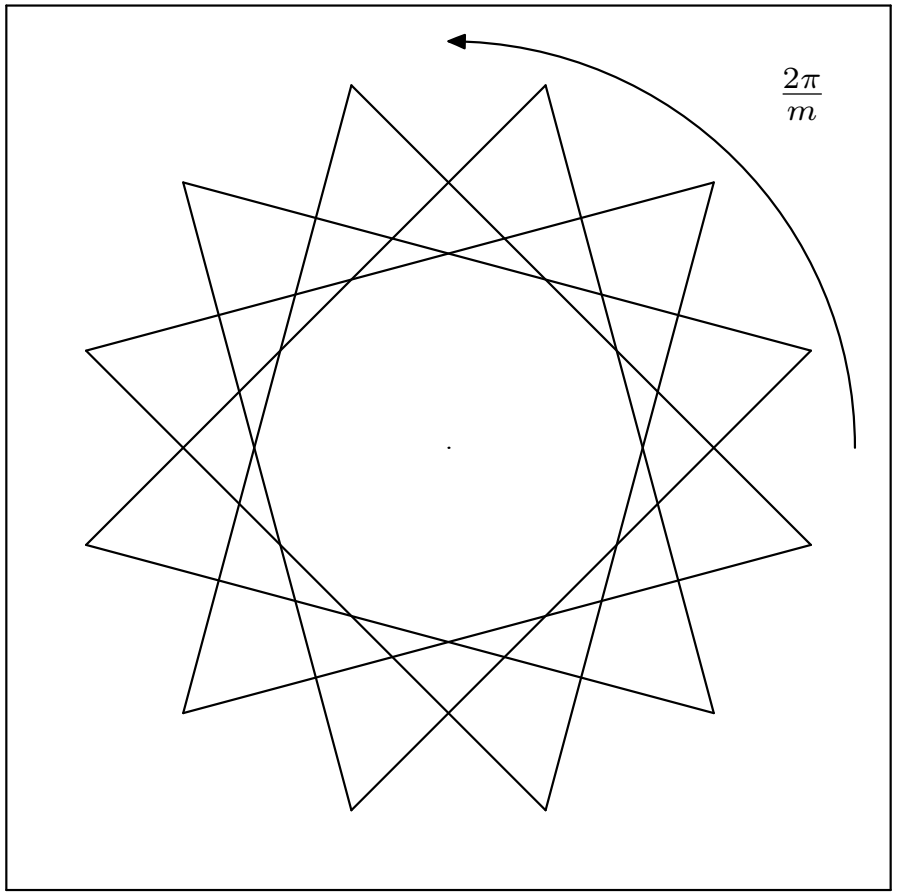
J. Regular Cake

First, compute the radius of the circumscribed circle of the inner polygon.



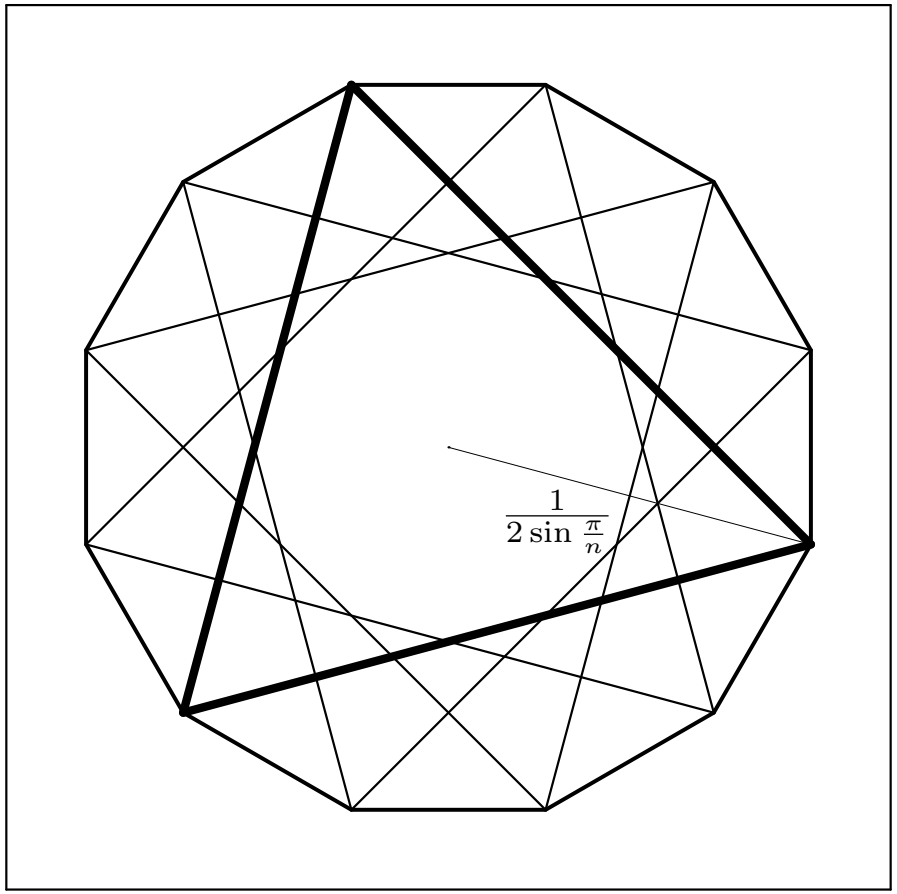
J. Regular Cake

Note that if the whole picture is rotated by $2\pi/m$ any number of times, the outer polygon stays the same.



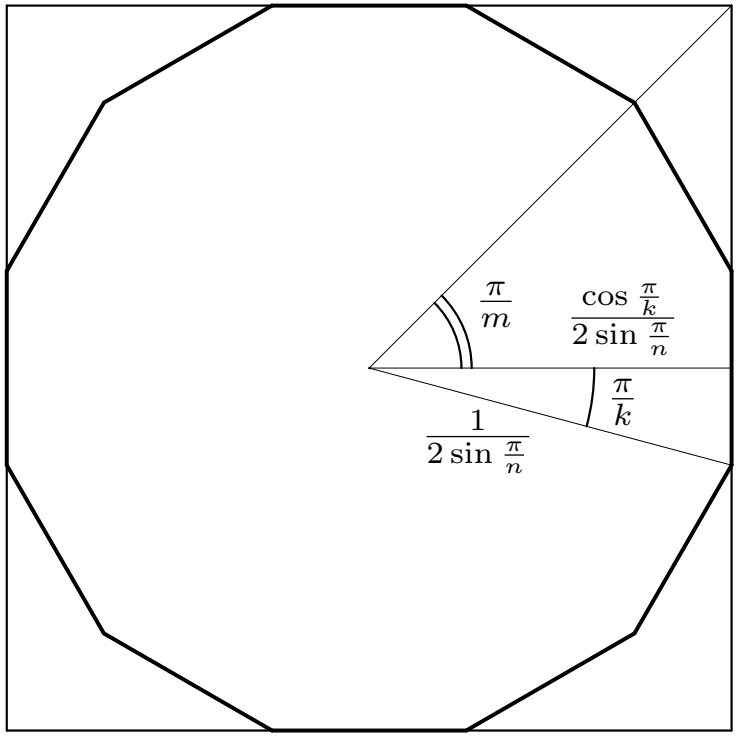
J. Regular Cake

However, the vertices of the inner polygon range equally spaced over the circumscribed circle. It is easy to see that the resulting shape is a regular k-gon, where $k = LCM(n, m)$.



J. Regular Cake

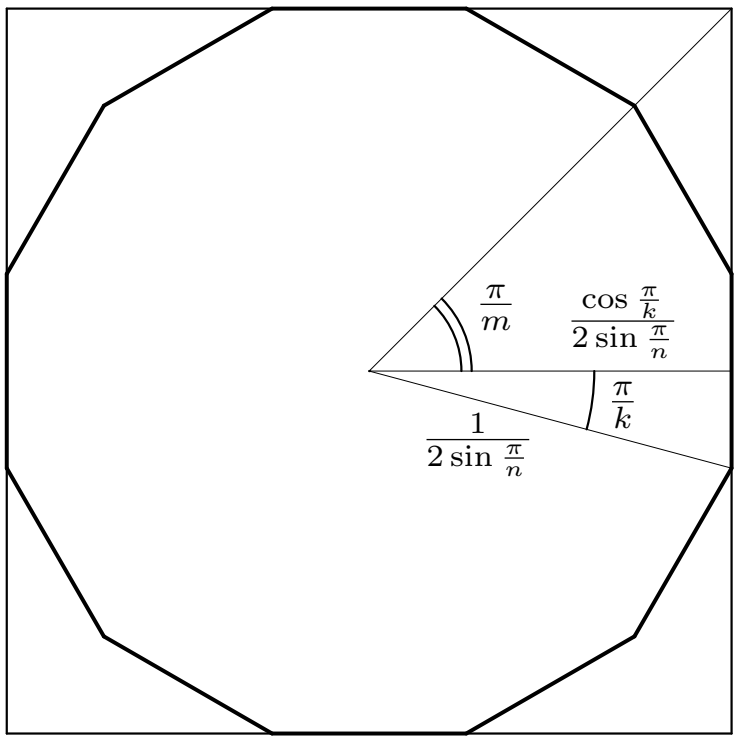
The inscribed circle of the m-gon cannot be smaller than of k-gon, but can be made equal by aligning sides since m divides k. The rest is straightforward trigonometry calculation.



$$\frac{\cos\frac{\pi}{k}}{2\sin\frac{\pi}{n}}\tan\frac{\pi}{m}$$

J. Regular Cake

The inscribed circle of the m-gon cannot be smaller than of k-gon, but can be made equal by aligning sides since m divides k. The rest is straightforward trigonometry calculation.



$\frac{\cos\frac{\pi}{k}}{2\sin\frac{\pi}{n}}\tan\frac{\pi}{m}$

The answer is $\frac{\cos(\pi/LCM(n,m))\tan(\pi/m)}{\sin(\pi/n)}$.

K. Piecemaking

Problem idea, developer: Alexander Ostanin
Editorial: Mikhail Tikhomirov

There is a tree with n vertices and $n - 1$ weighted edges. Some of the vertices are colored with black or white. Remove a set of edges with smallest total weight so that no black vertex is reachable from a white vertex.

K. Piecemaking

We will solve the problem via subtree DP. Let $dp_{v,f}$ be the smallest weight of removed edges in the subtree of v so that no conflicts occur, and, additionally:

K. Piecemaking

We will solve the problem via subtree DP. Let $dp_{v,f}$ be the smallest weight of removed edges in the subtree of v so that no conflicts occur, and, additionally:

- $f = 0$: no colored vertices are reachable from v in the subtree,

K. Piecemaking

We will solve the problem via subtree DP. Let $dp_{v,f}$ be the smallest weight of removed edges in the subtree of v so that no conflicts occur, and, additionally:

- $f = 0$: no colored vertices are reachable from v in the subtree,
- $f = 1$: only white vertices are reachable from v;

K. Piecemaking

We will solve the problem via subtree DP. Let $dp_{v,f}$ be the smallest weight of removed edges in the subtree of v so that no conflicts occur, and, additionally:

- $f = 0$: no colored vertices are reachable from v in the subtree,
- $f = 1$: only white vertices are reachable from v;
- $f = 2$: only black vertices are reachable from v.

K. Piecemaking

To recalculate $dp_{v,f}$ we process children one by one. Initially $dp_{v,0} = 0$, $dp_{v,1} = dp_{v,2} = \infty$ if v is uncolored, otherwise one of $dp_{v,1}$ and $dp_{v,2}$ is equal to 0.

K. Piecemaking

To recalculate $dp_{v,f}$ we process children one by one. Initially $dp_{v,0} = 0$, $dp_{v,1} = dp_{v,2} = \infty$ if v is uncolored, otherwise one of $dp_{v,1}$ and $dp_{v,2}$ is equal to 0.

If u is the new child, we use its DP values to compute $dp'_{v,f}$ — updated values of $dp_{v,f}$.

K. Piecemaking

To recalculate $dp_{v,f}$ we process children one by one. Initially $dp_{v,0} = 0$, $dp_{v,1} = dp_{v,2} = \infty$ if v is uncolored, otherwise one of $dp_{v,1}$ and $dp_{v,2}$ is equal to 0.

If u is the new child, we use its DP values to compute $dp'_{v,f}$ — updated values of $dp_{v,f}$.

We should consider all options for situations in both parts of the tree, and also whether we cut the edge or not. Each of these options may lead or not lead to a conflict depending on whether vertices of different color become exposed to each other or not. The cost is comprised of costs in two halves and (possibly) the cost of the recently removed edge.

K. Piecemaking

To recalculate $dp_{v,f}$ we process children one by one. Initially $dp_{v,0} = 0$, $dp_{v,1} = dp_{v,2} = \infty$ if v is uncolored, otherwise one of $dp_{v,1}$ and $dp_{v,2}$ is equal to 0.

If u is the new child, we use its DP values to compute $dp'_{v,f}$ — updated values of $dp_{v,f}$.

We should consider all options for situations in both parts of the tree, and also whether we cut the edge or not. Each of these options may lead or not lead to a conflict depending on whether vertices of different color become exposed to each other or not. The cost is comprised of costs in two halves and (possibly) the cost of the recently removed edge.

The answer is $\min_f dp_{r,f}$, where r is the root of the tree. In total, we obtain an $O(n)$ solution.

Thank you for reading!

Moscow Pre-Finals
ACM ICPC Workshop
★2017★