CCPC Regional Contest Editorial

November 16th, 2016

by Mikhail Tikhomirov (MIPT)

Moscow ACM ICPC Workshop, MIPT, 2016

A. Hanso vs Genji

A cylinder is flying in 3D space with an initial velocity under constant acceleration force. The axis of the cylinder is always aligned with the speed vector. Determine if the cylinder will ever hit a given point q.

A. Hanso vs Genji

A cylinder is flying in 3D space with an initial velocity under constant acceleration force. The axis of the cylinder is always aligned with the speed vector. Determine if the cylinder will ever hit a given point q.

Outline: reduce to a 2D problem, solve some polynomial inequalities.

A. Hanso vs Genji

Let v be the initial speed vector, and g be the gravitational force. Consider π — the plane containing the initial position of mass center, and parallel to v and g. Clearly, the mass center will always stay inside this plane. It is also evident that the distance from any fixed point of the cylinder to this plane will remain constant.

A. Hanso vs Genji

Let v be the initial speed vector, and g be the gravitational force. Consider π — the plane containing the initial position of mass center, and parallel to v and g. Clearly, the mass center will always stay inside this plane. It is also evident that the distance from any fixed point of the cylinder to this plane will remain constant.

Let us cut the cylinder with a plane parallel to π that passes through q. The intersection is either empty or a rectangle. It is this rectangle that has the change to hit the point q. Also note that it behaves like a full 2D analogue of the cylinder (if we take its center to be the mass center). We can thus reduce to a 2D version of the problem. Introduce a 2D coordinate system in a suitable way.

A. Hanso vs Genji

Let the center mass move according to the rule $p(x) = (x, v_y x - gx^2/2)$, the speed vector is then equal to $v(x) = (1, v_y - gx)$. Also let $u(x) = (v_y - gx, -1)$ be the vector orthogonal to v.

Assume that the rectangle's speed-aligned side's length is equal to L, and the orthogonal side's length is equal to W.

A. Hanso vs Genji

Let the center mass move according to the rule $p(x) = (x, v_y x - gx^2/2)$, the speed vector is then equal to $v(x) = (1, v_y - gx)$. Also let $u(x) = (v_y - gx, -1)$ be the vector orthogonal to v.

Assume that the rectangle's speed-aligned side's length is equal to L, and the orthogonal side's length is equal to W.

The rectangle contains a point q at time moment x iff

$$\left|\left(q - p(x), \frac{v(x)}{|v(x)|}\right)\right| \leqslant L, \left|\left(q - p(x), \frac{u(x)}{|u(x)|}\right)\right| \leqslant W.$$

Here $(\cdot, \cdot)$ stands for dot product.

A. Hanso vs Genji

Let the center mass move according to the rule $p(x) = (x, v_y x - gx^2/2)$, the speed vector is then equal to $v(x) = (1, v_y - gx)$. Also let $u(x) = (v_y - gx, -1)$ be the vector orthogonal to v.

Assume that the rectangle's speed-aligned side's length is equal to L, and the orthogonal side's length is equal to W.

The rectangle contains a point q at time moment x iff

$$\left|\left(q - p(x), \frac{v(x)}{|v(x)|}\right)\right| \leqslant L, \left|\left(q - p(x), \frac{u(x)}{|u(x)|}\right)\right| \leqslant W.$$

Here $(\cdot, \cdot)$ stands for dot product.

Consider the first inequality (the second is almost analogous). After squaring we obtain an equivalent form:

$$(q - p(x), v(x))^2 \leqslant L^2 |v(x)|^2.$$

All vector coordinates are polynomials in x. It can be checked that this is a polynomial inequality of degree 6.

A. Hanso vs Genji

How does one find the solution domain of an arbitrary polynomial inequality? We will describe a method for finding roots of an arbitrary polynomial; the method can be upgraded for finding the domain too.

A. Hanso vs Genji

How does one find the solution domain of an arbitrary polynomial inequality? We will describe a method for finding roots of an arbitrary polynomial; the method can be upgraded for finding the domain too.

Let us have $P(x) = 0$.

A. Hanso vs Genji

How does one find the solution domain of an arbitrary polynomial inequality? We will describe a method for finding roots of an arbitrary polynomial; the method can be upgraded for finding the domain too.

Let us have $P(x) = 0$.

Recursively solve for $P'(x) = 0$, where P' is a formal derivative of P. There is at most one root of P between two consecutive roots of P', use binary search to find or discard them.

A. Hanso vs Genji

How does one find the solution domain of an arbitrary polynomial inequality? We will describe a method for finding roots of an arbitrary polynomial; the method can be upgraded for finding the domain too.

Let us have $P(x) = 0$.

Recursively solve for $P'(x) = 0$, where P' is a formal derivative of P. There is at most one root of P between two consecutive roots of P', use binary search to find or discard them.

Once we have $\deg P = 1$, solving the equation is trivial.

A. Hanso vs Genji

How does one find the solution domain of an arbitrary polynomial inequality? We will describe a method for finding roots of an arbitrary polynomial; the method can be upgraded for finding the domain too.

Let us have $P(x) = 0$.

Recursively solve for $P'(x) = 0$, where P' is a formal derivative of P. There is at most one root of P between two consecutive roots of P', use binary search to find or discard them.

Once we have $\deg P = 1$, solving the equation is trivial.

To obtain the solution we have to intersect the domains for two inequalities. Each of them is a union of several segments, thus the task is straightforward.

A. Hanso vs Genji

How does one find the solution domain of an arbitrary polynomial inequality? We will describe a method for finding roots of an arbitrary polynomial; the method can be upgraded for finding the domain too.

Let us have $P(x) = 0$.

Recursively solve for $P'(x) = 0$, where P' is a formal derivative of P. There is at most one root of P between two consecutive roots of P', use binary search to find or discard them.

Once we have $\deg P = 1$, solving the equation is trivial.

To obtain the solution we have to intersect the domains for two inequalities. Each of them is a union of several segments, thus the task is straightforward.

Might take several tries for precision issues if the moon phase is wrong.

A. Hanso vs Genji

How does one find the solution domain of an arbitrary polynomial inequality? We will describe a method for finding roots of an arbitrary polynomial; the method can be upgraded for finding the domain too.

Let us have $P(x) = 0$.

Recursively solve for $P'(x) = 0$, where P' is a formal derivative of P. There is at most one root of P between two consecutive roots of P', use binary search to find or discard them.

Once we have $\deg P = 1$, solving the equation is trivial.

To obtain the solution we have to intersect the domains for two inequalities. Each of them is a union of several segments, thus the task is straightforward.

Might take several tries for precision issues if the moon phase is wrong.

A number of other numerical approaches is available. May have different odds to pass depending on details.

B. Fraction

Compute the value of chain fraction as a reduced rational number.

B. Fraction

Compute the value of chain fraction as a reduced rational number.

Outline: just do the basic fractions maniplations and reduce by GCD.

B. Fraction

Compute the value of chain fraction as a reduced rational number.

Outline: just do the basic fractions maniplations and reduce by GCD.

The numbers were so small you could even go with `int`'s for values.

C. Rotate string

We call a string *representative* if it's equal to its least cyclic shift. Given a string s, find the number of representative strings that are lexicographically smaller or equal to s.

C. Rotate string

We call a string *representative* if it's equal to its least cyclic shift. Given a string s, find the number of representative strings that are lexicographically smaller or equal to s.

Outline: count the number of "bad" strings such that all their cyclic shifts are greater than s. Use inclusion-exclusion to account for periodicity.

C. Rotate string

An algorithm for checking if all *substrings* of t are greater than corresponding prefixes of s:

C. Rotate string

An algorithm for checking if all *substrings* of t are greater than corresponding prefixes of s:

- For current position i store maximal l such that $t[i-l+1..i] = s[1..i]$.

C. Rotate string

An algorithm for checking if all *substrings* of t are greater than corresponding prefixes of s:

- For current position i store maximal l such that $t[i - l + 1..i] = s[1..i]$.
- To append a single character c, repeatedly apply prefix-function of s to i. On each iteration ensure that continuation of the substring that matches until i does not fall under the prefix of s.

C. Rotate string

An algorithm for checking if all *substrings* of t are greater than corresponding prefixes of s:

- For current position i store maximal l such that $t[i-l+1..i] = s[1..i]$.
- To append a single character c, repeatedly apply prefix-function of s to i. On each iteration ensure that continuation of the substring that matches until i does not fall under the prefix of s.

How do we apply this idea to check the same property for cyclic shifts? The only difference is that now we assume that at the start l is the maximal suffix of t that matches prefix of s. In the end l has to come to its original value.

C. Rotate string

We should count the "prefix-function automaton" that can append a symbol to a string t and know what will the maximum prefix-suffix match will be; it will also forbid the transitions that allow a substring to fall under a prefix of s lexicographically.

C. Rotate string

We should count the “prefix-function automaton” that can append a symbol to a string t and know what will the maximum prefix-suffix match will be; it will also forbid the transitions that allow a substring to fall under a prefix of s lexicographically.

We will now count the “bad” strings according to the algorithm above. Fix the original value of l. The DP will store the number of processed symbols as well the current value of l. All ways to transfer from l in the beginning to l in the end will correspond to bad strings.

C. Rotate string

We should count the “prefix-function automaton” that can append a symbol to a string t and know what will the maximum prefix-suffix match will be; it will also forbid the transitions that allow a substring to fall under a prefix of s lexicographically.

We will now count the “bad” strings according to the algorithm above. Fix the original value of l. The DP will store the number of processed symbols as well the current value of l. All ways to transfer from l in the beginning to l in the end will correspond to bad strings.

All the rest $(26^n - x)$ strings have at least one cyclic shift that falls under s lexicographically. However, strings with period d will be counted d times. To mitigate this, use the standard inclusion-exclusion method for divisors of n.

D. Triangle

Find the largest subset of $\{1, \ldots, n\}$ that doesn't contain lengths of three sides of a triangle.

D. Triangle

Find the largest subset of $\{1, \dots, n\}$ that doesn't contain lengths of three sides of a triangle.

Outline: construct the set greedily in $O(\log n)$ time.

D. Triangle

Let a and b be the two smallest element of the set. Clearly, the third smallest element will be at least $a + b$, the fourth will be at least $b + (a + b)$, and so on.

D. Triangle

Let a and b be the two smallest element of the set. Clearly, the third smallest element will be at least $a + b$, the fourth will be at least $b + (a + b)$, and so on.

If we want to fit the most elements in $\{1, \ldots, n\}$ we should choose the smallest possible number every time.

D. Triangle

Let a and b be the two smallest element of the set. Clearly, the third smallest element will be at least $a + b$, the fourth will be at least $b + (a + b)$, and so on.

If we want to fit the most elements in $\{1, \ldots, n\}$ we should choose the smallest possible number every time.

That results in a shifted Fibonacci sequence: $1, 2, 3, 5, 8, \ldots$.

D. Triangle

Let a and b be the two smallest element of the set. Clearly, the third smallest element will be at least $a + b$, the fourth will be at least $b + (a + b)$, and so on.

If we want to fit the most elements in $\{1, \ldots, n\}$ we should choose the smallest possible number every time.

That results in a shifted Fibonacci sequence: $1, 2, 3, 5, 8, \ldots$.

Since it grows exponentially, it takes $O(\log n)$ to count the number of elements not greater than n with straightforward computation.

E. The Fastest Runner Ms. Zhang

In a connected graph with n vertices and n edges find the minimum length of a path that visits each vertex.

E. The Fastest Runner Ms. Zhang

In a connected graph with n vertices and n edges find the minimum length of a path that visits each vertex.

Outline: try to delete each edge in the cycle and solve the problem for the remaining tree in each case. Optimize with some data structures.

E. The Fastest Runner Ms. Zhang

Let's first solve the problem if the graph is a tree.

E. The Fastest Runner Ms. Zhang

Let's first solve the problem if the graph is a tree.

For a particular pair of start and finish vertex v, u the answer is at least $2(n-1) - d(v, u)$, where $d(v, u)$ is the distance between v and u.

E. The Fastest Runner Ms. Zhang

Let's first solve the problem if the graph is a tree.

For a particular pair of start and finish vertex v, u the answer is at least $2(n-1) - d(v, u)$, where $d(v, u)$ is the distance between v and u.

Indeed, if an edge doesn't lie on the shortest $v - u$ path, we have to traverse it at least twice since we have to visit vertices on the other side, but both v and u lie on the same side of the edge so we have to return.

E. The Fastest Runner Ms. Zhang

Let's first solve the problem if the graph is a tree.

For a particular pair of start and finish vertex v, u the answer is at least $2(n-1) - d(v, u)$, where $d(v, u)$ is the distance between v and u.

Indeed, if an edge doesn't lie on the shortest $v - u$ path, we have to traverse it at least twice since we have to visit vertices on the other side, but both v and u lie on the same side of the edge so we have to return.

If the edge lies on the $v - u$ path, we have to traverse it at least once to get from v to u.

E. The Fastest Runner Ms. Zhang

Let's first solve the problem if the graph is a tree.

For a particular pair of start and finish vertex v, u the answer is at least $2(n-1) - d(v,u)$, where $d(v,u)$ is the distance between v and u.

Indeed, if an edge doesn't lie on the shortest $v - u$ path, we have to traverse it at least twice since we have to visit vertices on the other side, but both v and u lie on the same side of the edge so we have to return.

If the edge lies on the $v - u$ path, we have to traverse it at least once to get from v to u.

This length is attained on a "partial" Euler tour of the tree, so this must be the answer for the choice of v and u.

E. The Fastest Runner Ms. Zhang

Let's first solve the problem if the graph is a tree.

For a particular pair of start and finish vertex v, u the answer is at least $2(n-1) - d(v, u)$, where $d(v, u)$ is the distance between v and u.

Indeed, if an edge doesn't lie on the shortest $v - u$ path, we have to traverse it at least twice since we have to visit vertices on the other side, but both v and u lie on the same side of the edge so we have to return.

If the edge lies on the $v - u$ path, we have to traverse it at least once to get from v to u.

This length is attained on a "partial" Euler tour of the tree, so this must be the answer for the choice of v and u.

To optimize the length, we have to choose v and u as endpoints of a diameter of the tree.

E. The Fastest Runner Ms. Zhang

In the actual problem we have a graph that consists of exactly one cycle with some trees hanging from each cycle vertex.

E. The Fastest Runner Ms. Zhang

In the actual problem we have a graph that consists of exactly one cycle with some trees hanging from each cycle vertex.

~~It doesn't make sense to travel each edge of the cycle.~~ *Actually, it does when we start in a subtree, rise up to the cycle, make a whole loop visiting all subtrees while we go, and then return to the same subtree we started. All these options can be accounted for in linear time, and all the other routes do not need to visit all cycle edges indeed.* With this observation we can obtain an easy solution: first find the cycle in the graph, then try to erase each edge of the cycle and apply the solution to the remaining tree. There could be $\sim n$ options to try though.

E. The Fastest Runner Ms. Zhang

Denote $v_1, \ldots, v_k$ be the vertices of the cycle in order. Consider a tree that hangs on v_i. Let it have the diameter d_i and the longest path down from the root v_i have length l_i.

E. The Fastest Runner Ms. Zhang

Denote $v_1, \ldots, v_k$ be the vertices of the cycle in order. Consider a tree that hangs on v_i. Let it have the diameter d_i and the longest path down from the root v_i have length l_i.

If we cut the (v_k, v_1) edge, the diameter of the resulting tree is

$$\max\left(\max_{i=1}^{k} d_i, \max_{1 \leqslant i < j \leqslant k} l_i + j - i + l_j\right)$$

The second part of the expression corresponds to all options to draw a path between different trees.

E. The Fastest Runner Ms. Zhang

Denote $v_1, \ldots, v_k$ be the vertices of the cycle in order. Consider a tree that hangs on v_i. Let it have the diameter d_i and the longest path down from the root v_i have length l_i.

If we cut the (v_k, v_1) edge, the diameter of the resulting tree is

$$\max\left(\max_{i=1}^{k} d_i, \max_{1\leqslant i<j\leqslant k} l_i + j - i + l_j\right)$$

The second part of the expression corresponds to all options to draw a path between different trees.

Note that this expression can be computed in $O(n)$ time. To find the second part, we have to try all j and choose $i < j$ that maximizes $l_i - i$.

E. The Fastest Runner Ms. Zhang

Denote $v_1, \ldots, v_k$ be the vertices of the cycle in order. Consider a tree that hangs on v_i. Let it have the diameter d_i and the longest path down from the root v_i have length l_i.

If we cut the (v_k, v_1) edge, the diameter of the resulting tree is

$$\max\left(\max_{i=1}^{k} d_i, \max_{1 \leqslant i < j \leqslant k} l_i + j - i + l_j\right)$$

The second part of the expression corresponds to all options to draw a path between different trees.

Note that this expression can be computed in $O(n)$ time. To find the second part, we have to try all j and choose $i < j$ that maximizes $l_i - i$.

To do that fast we store the maximal value of $l_i - i$ over all processed j.

E. The Fastest Runner Ms. Zhang

How to account for all possible ways to erase an edge? Let us double the array l_i, that is, append the same elements at the end: $(l_1, \ldots, l_k, l_1, \ldots, l_k)$.

E. The Fastest Runner Ms. Zhang

How to account for all possible ways to erase an edge? Let us double the array l_i, that is, append the same elements at the end: $(l_1, \ldots, l_k, l_1, \ldots, l_k)$.

The new array of length $2k$ contains all cyclic shifts of the original array.

E. The Fastest Runner Ms. Zhang

How to account for all possible ways to erase an edge? Let us double the array l_i, that is, append the same elements at the end: $(l_1, \dots, l_k, l_1, \dots, l_k)$.

The new array of length $2k$ contains all cyclic shifts of the original array.

The maximal diameter of all possible trees is now either the maximal d_i for a certain i or $l_j + j + l_i - i$ for a certain pair $1 \leqslant i < j \leqslant 2k$ that satisfies $j - i < n$.

E. The Fastest Runner Ms. Zhang

How to account for all possible ways to erase an edge? Let us double the array l_i, that is, append the same elements at the end: $(l_1, \ldots, l_k, l_1, \ldots, l_k)$.

The new array of length $2k$ contains all cyclic shifts of the original array.

The maximal diameter of all possible trees is now either the maximal d_i for a certain i or $l_j + j + l_i - i$ for a certain pair $1 \leqslant i < j \leqslant 2k$ that satisfies $j - i < n$.

Finding a maximal i for each j now looks like an RMQ instance. We can solve it with any RMQ structure or an `std::set`+two pointers since we know all the queries from the start, and both ends of the segments are monotonous.

E. The Fastest Runner Ms. Zhang

How to account for all possible ways to erase an edge? Let us double the array l_i, that is, append the same elements at the end: $(l_1, \ldots, l_k, l_1, \ldots, l_k)$.

The new array of length $2k$ contains all cyclic shifts of the original array.

The maximal diameter of all possible trees is now either the maximal d_i for a certain i or $l_j + j + l_i - i$ for a certain pair $1 \leqslant i < j \leqslant 2k$ that satisfies $j - i < n$.

Finding a maximal i for each j now looks like an RMQ instance. We can solve it with any RMQ structure or an `std::set`+two pointers since we know all the queries from the start, and both ends of the segments are monotonous.

If the maximal diameter is D, then the answer is $2(n-1) - D$ as before.

E. The Fastest Runner Ms. Zhang

How to account for all possible ways to erase an edge? Let us double the array l_i, that is, append the same elements at the end: $(l_1, \ldots, l_k, l_1, \ldots, l_k)$.

The new array of length $2k$ contains all cyclic shifts of the original array.

The maximal diameter of all possible trees is now either the maximal d_i for a certain i or $l_j + j + l_i - i$ for a certain pair $1 \leqslant i < j \leqslant 2k$ that satisfies $j - i < n$.

Finding a maximal i for each j now looks like an RMQ instance. We can solve it with any RMQ structure or an `std::set`+two pointers since we know all the queries from the start, and both ends of the segments are monotonous.

If the maximal diameter is D, then the answer is $2(n-1) - D$ as before.

The total complexity is $O(n \log n)$.

F. Harmonic Value

Harmonic value of a permutation $(p_1, \ldots, p_n)$ is the sum $f(p) = \sum_{i=1}^{n-1} \mathrm{GCD}(p_i, p_{i+1})$. Find the k-th smallest possible value of harmonic sum of a permutation of n numbers and present a permutation with such value. $2k \leqslant n$.

F. Harmonic Value

Harmonic value of a permutation $(p_1, \ldots, p_n)$ is the sum $f(p) = \sum_{i=1}^{n-1} \mathrm{GCD}(p_i, p_{i+1})$. Find the k-th smallest possible value of harmonic sum of a permutation of n numbers and present a permutation with such value. $2k \leqslant n$.

Outline: for $2k \leqslant n$ a really simple construction works. Without this condition — hard.

F. Harmonic Value

Clearly, $f(p) \geqslant n - 1$. Let us present a construction with a single value of GCD different from 1 being k (only for $2k \leqslant n$).

F. Harmonic Value

Clearly, $f(p) \geqslant n-1$. Let us present a construction with a single value of GCD different from 1 being k (only for $2k \leqslant n$).

If k is even: $1, 2, \ldots, k, 2k, 2k-1, \ldots, k+1, 2k+1, 2k+2, \ldots, n$.

F. Harmonic Value

Clearly, $f(p) \geqslant n-1$. Let us present a construction with a single value of GCD different from 1 being k (only for $2k \leqslant n$).

If k is even: $1, 2, \ldots, k, 2k, 2k-1, \ldots, k+1, 2k+1, 2k+2, \ldots, n$.

It's easy to check that each of the adjacent pairs other than $(k, 2k)$ either differs by 1 or is a pair $(k+1, 2k+1)$ which has GCD of 1.

F. Harmonic Value

Clearly, $f(p) \geqslant n-1$. Let us present a construction with a single value of GCD different from 1 being k (only for $2k \leqslant n$).

If k is even: $1, 2, \ldots, k, 2k, 2k-1, \ldots, k+1, 2k+1, 2k+2, \ldots, n$.

It's easy to check that each of the adjacent pairs other than $(k, 2k)$ either differs by 1 or is a pair $(k+1, 2k+1)$ which has GCD of 1.

If k is odd: $1, 2, \ldots, k, 2k, k+1, \ldots, 2k-1, 2k+1, 2k+2, \ldots, n$.

F. Harmonic Value

Clearly, $f(p) \geqslant n-1$. Let us present a construction with a single value of GCD different from 1 being k (only for $2k \leqslant n$).

If k is even: $1, 2, \ldots, k, 2k, 2k-1, \ldots, k+1, 2k+1, 2k+2, \ldots, n$.

It's easy to check that each of the adjacent pairs other than $(k, 2k)$ either differs by 1 or is a pair $(k+1, 2k+1)$ which has GCD of 1.

If k is odd: $1, 2, \ldots, k, 2k, k+1, \ldots, 2k-1, 2k+1, 2k+2, \ldots, n$.

Again, adjacent pairs either differ by 1, or are $(k, 2k)$ or $(2k-1, 2k+1)$.

F. Harmonic Value

Clearly, $f(p) \geqslant n-1$. Let us present a construction with a single value of GCD different from 1 being k (only for $2k \leqslant n$).

If k is even: $1,\ 2,\ \ldots,\ k,\ 2k,\ 2k-1,\ \ldots,\ k+1,\ 2k+1,\ 2k+2,\ \ldots,\ n$.

It's easy to check that each of the adjacent pairs other than $(k, 2k)$ either differs by 1 or is a pair $(k+1, 2k+1)$ which has GCD of 1.

If k is odd: $1,\ 2,\ \ldots,\ k,\ 2k,\ k+1,\ \ldots,\ 2k-1,\ 2k+1,\ 2k+2,\ \ldots,\ n$.

Again, adjacent pairs either differ by 1, or are $(k, 2k)$ or $(2k-1, 2k+1)$.

Evidently enough from above, the k-th smallest value of f is $n-k+2$.

G. Instability

In a given graph count the number of (induced) subgraphs that contain either a triangle or an anti-triangle.

G. Instability

In a given graph count the number of (induced) subgraphs that contain either a triangle or an anti-triangle.

Outline: all subsets with six or more vertices are always good, all the others can be brute-forced.

G. Instability

Theorem

Every graph of $n \geqslant 6$ contains either or a triangle or an anti-triangle.

G. Instability

Theorem

Every graph of $n \geqslant 6$ contains either or a triangle or an anti-triangle.

Proof.

A classic exercise in graph theory. $\square$

G. Instability

Theorem

Every graph of $n \geqslant 6$ contains either or a triangle or an anti-triangle.

Proof.

A classic exercise in graph theory. $\square$

In general:

Ramsey's theorem

For each $r, s > 0$ there is such n that every graph on at least n vertices contains either an r-clique or an s-anticlique.

G. Instability

The theorem above implies that all the subsets of size at least 6 should be included in the answer. It suffices to check all subsets of size at most 5 in $O(n^5)$ time.

G. Instability

The theorem above implies that all the subsets of size at least 6 should be included in the answer. It suffices to check all subsets of size at most 5 in $O(n^5)$ time.

While the complexity may seem large, remember that the constant is effectively $1/5!$. Also various tricks may be employed to further optimize the solution.

H. Sequence I

For two sequences a and b and a number p, count the number of subsequences of a with distance p between successive indices that are equal to b.

H. Sequence I

For two sequences a and b and a number p, count the number of subsequences of a with distance p between successive indices that are equal to b.

Outline: simple reduction to substring search.

H. Sequence I

For $1 \leqslant i \leqslant p$ consider a sequence $c_i = (a_i, a_{i+p}, \ldots)$. We can count the number of substrings of c_i that are equal to b in $O(|b| + |c_i|)$ time with any substring search algorithm, e.g., KMP.

H. Sequence I

For $1 \leqslant i \leqslant p$ consider a sequence $c_i = (a_i, a_{i+p}, \ldots)$. We can count the number of substrings of c_i that are equal to b in $O(|b| + |c_i|)$ time with any substring search algorithm, e.g., KMP.

Doing this for all i results in a $O(|a| + p|b|)$ time solution.

H. Sequence I

For $1 \leqslant i \leqslant p$ consider a sequence $c_i = (a_i, a_{i+p}, \ldots)$. We can count the number of substrings of c_i that are equal to b in $O(|b| + |c_i|)$ time with any substring search algorithm, e.g., KMP.

Doing this for all i results in a $O(|a| + p|b|)$ time solution.

Don't do anything for particular i if $|c_i| < |b|$: $O(|a| + |b|)$ time.

I. Sequence II

We are given an array of integers $a_1, \dots, a_n$. For a segment $[l; r]$ call a position i *interesting* if it's the first occurence of the number a_i in the segment. Process several queries "find median of all interesting positions of segment $[l_i; r_i]$". Queries must be answered *online*.

I. Sequence II

We are given an array of integers $a_1, \ldots, a_n$. For a segment $[l; r]$ call a position i *interesting* if it's the first occurence of the number a_i in the segment. Process several queries "find median of all interesting positions of segment $[l_i; r_i]$". Queries must be answered *online*.

Outline: resort to cartesian trees for queries, make them persistent to make the solution online.

I. Sequence II

It usually helps to come up with an offline solution first. Let's process all queries by decreasing of l_i. We will store the set of all interesting positions in the segment $[l; n]$.

I. Sequence II

It usually helps to come up with an offline solution first. Let's process all queries by decreasing of l_i. We will store the set of all interesting positions in the segment $[l; n]$.

When l is decreased, a new interesting position l is created, possibly overwriting a previous interesting position if a_l is encountered later in the array.

I. Sequence II

It usually helps to come up with an offline solution first. Let's process all queries by decreasing of l_i. We will store the set of all interesting positions in the segment $[l; n]$.

When l is decreased, a new interesting position l is created, possibly overwriting a previous interesting position if a_l is encountered later in the array.

Let us store all interesting positions in a cartesian tree. To answer a query $[l; r]$, perform a cut of the tree in position r. We will then know the number of interesting positions and will be able to address a specific index.

I. Sequence II

It usually helps to come up with an offline solution first. Let's process all queries by decreasing of l_i. We will store the set of all interesting positions in the segment $[l; n]$.

When l is decreased, a new interesting position l is created, possibly overwriting a previous interesting position if a_l is encountered later in the array.

Let us store all interesting positions in a cartesian tree. To answer a query $[l; r]$, perform a cut of the tree in position r. We will then know the number of interesting positions and will be able to address a specific index.

Also note that we can avoid actually modifying the tree just by performing a "binary-search" descent and counting the number of elements $\leqslant r$.

I. Sequence II

It usually helps to come up with an offline solution first. Let's process all queries by decreasing of l_i. We will store the set of all interesting positions in the segment $[l; n]$.

When l is decreased, a new interesting position l is created, possibly overwriting a previous interesting position if a_l is encountered later in the array.

Let us store all interesting positions in a cartesian tree. To answer a query $[l; r]$, perform a cut of the tree in position r. We will then know the number of interesting positions and will be able to address a specific index.

Also note that we can avoid actually modifying the tree just by performing a "binary-search" descent and counting the number of elements $\leqslant r$.

This solution is now $O(\log n)$ per query and $O(n \log n)$ preprocessing.

I. Sequence II

To answer queries online, make the tree persistent; this will allow us to access any “version” of the tree and answer any query after the preprocessing.

I. Sequence II

To answer queries online, make the tree persistent; this will allow us to access any “version” of the tree and answer any query after the preprocessing.

This will not affect the time complexity, but will require $O(n \log n)$ memory.

J. Ugly Problem

Represent $n \leqslant 10^{1000}$ as a sum of at most 50 palindrome numbers.

J. Ugly Problem

Represent $n \leqslant 10^{1000}$ as a sum of at most 50 palindrome numbers.

Outline: greedily subtracting largest possible palindrome roughly halves the length of the number.

J. Ugly Problem

We want to subtract the largest palindrome not exceeding n.

J. Ugly Problem

We want to subtract the largest palindrome not exceeding n.

Let us suppose that the length of n is l. We will build m as follows: take $l' = \lceil l/2 \rceil$ first digits of n and add the rest so that m has l digits and is a palindrome.

J. Ugly Problem

We want to subtract the largest palindrome not exceeding n.

Let us suppose that the length of n is l. We will build m as follows: take $l' = \lceil l/2 \rceil$ first digits of n and add the rest so that m has l digits and is a palindrome.

If $m > n$, subtract 1 from the number formed by the l' largest digits of m and mirror the number again; the new number is less than n. Now we can subtract m from n.

J. Ugly Problem

We want to subtract the largest palindrome not exceeding n.

Let us suppose that the length of n is l. We will build m as follows: take $l' = \lceil l/2 \rceil$ first digits of n and add the rest so that m has l digits and is a palindrome.

If $m > n$, subtract 1 from the number formed by the l' largest digits of m and mirror the number again; the new number is less than n. Now we can subtract m from n.

The special case is when $n = 10^k$, then we need to take $m = 10^k - 1$.

J. Ugly Problem

We want to subtract the largest palindrome not exceeding n.

Let us suppose that the length of n is l. We will build m as follows: take $l' = \lceil l/2 \rceil$ first digits of n and add the rest so that m has l digits and is a palindrome.

If $m > n$, subtract 1 from the number formed by the l' largest digits of m and mirror the number again; the new number is less than n. Now we can subtract m from n.

The special case is when $n = 10^k$, then we need to take $m = 10^k - 1$.

If we don't need to alter m after making the first part equal, then $n - m < 10^{l'}$. In the other case, $n - m < 2 \cdot 10^{l'}$. In any case, then length of n is roughly halved after every operation, resulting in $O(\log n)$ summands.

K. Binary Indexed Tree

Count the total number of elements the BIT (Fenwick tree, etc.) performs for operations “change array elements at positions l and r” pairs (l, r) such that $0 \leqslant l < r \leqslant n$.

K. Binary Indexed Tree

Count the total number of elements the BIT (Fenwick tree, etc.) performs for operations “change array elements at positions l and r” pairs (l, r) such that $0 \leqslant l < r \leqslant n$.

Outline: look at binary representation of l and r, express the answer and find it combinatorially/with bitwise DP.

K. Binary Indexed Tree

The operation `i = i -= i & (-i)` effectively wipes the smallest bit of i set to 1.

K. Binary Indexed Tree

The operation `i = i -= i & (-i)` effectively wipes the smallest bit of i set to 1.

Let us call a number i a *prefix* of number j if i can be obtained from j using one or several operations described above. For a pair of numbers l and r the number of changed elements is the number of prefixes of l that are not prefixes of r, plus the symmetrical value.

K. Binary Indexed Tree

The operation `i = i -= i & (-i)` effectively wipes the smallest bit of i set to 1.

Let us call a number i a *prefix* of number j if i can be obtained from j using one or several operations described above. For a pair of numbers l and r the number of changed elements is the number of prefixes of l that are not prefixes of r, plus the symmetrical value.

A different way to express the answer: for each number x from 1 to n denote $f(x)$ the number of pairs (a, b) with $0 \leqslant a, b \leqslant n$ such that x is a prefix of a but not a prefix of b. Observe that $\sum_{x=1}^{n} f(x)$ differs from the actual answer only in the order of summation.

K. Binary Indexed Tree

Further, let $g(x)$ denote the number of y's not exceeding n such that x is a prefix of y, then $f(x) = g(x)(n + 1 - g(x))$.

K. Binary Indexed Tree

Further, let $g(x)$ denote the number of y's not exceeding n such that x is a prefix of y, then $f(x) = g(x)(n + 1 - g(x))$.

If $k(x)$ is the number of trailing zeros in binary representation of x, then $g(x) = n - x + 1$ if x is prefix of n, and $2^{k(x)}$ otherwise.

K. Binary Indexed Tree

Further, let $g(x)$ denote the number of y's not exceeding n such that x is a prefix of y, then $f(x) = g(x)(n + 1 - g(x))$.

If $k(x)$ is the number of trailing zeros in binary representation of x, then $g(x) = n - x + 1$ if x is prefix of n, and $2^{k(x)}$ otherwise.

For example, let $n = 13 = 1101_2$. Then
$g(4) = g(100_2) = 4 = |\{100_2, 101_2, 110_2, 111_2\}|$, but
$g(8) = g(1000_2) = 6 = |\{1000_2, 1001_2, 1010_2, 1011_2, 1100_2, 1101_2\}|$.

K. Binary Indexed Tree

Further, let $g(x)$ denote the number of y's not exceeding n such that x is a prefix of y, then $f(x) = g(x)(n + 1 - g(x))$.

If $k(x)$ is the number of trailing zeros in binary representation of x, then $g(x) = n - x + 1$ if x is prefix of n, and $2^{k(x)}$ otherwise.

For example, let $n = 13 = 1101_2$. Then
$g(4) = g(100_2) = 4 = |\{100_2, 101_2, 110_2, 111_2\}|$, but
$g(8) = g(1000_2) = 6 = |\{1000_2, 1001_2, 1010_2, 1011_2, 1100_2, 1101_2\}|$.

The number of x's with $k(x) = t$ is $\lfloor (n + 2^t)/2^{t+1} \rfloor$, since each of them is $(2z + 1)2^t$ for a non-negative integer z.

K. Binary Indexed Tree

Further, let $g(x)$ denote the number of y's not exceeding n such that x is a prefix of y, then $f(x) = g(x)(n + 1 - g(x))$.

If $k(x)$ is the number of trailing zeros in binary representation of x, then $g(x) = n - x + 1$ if x is prefix of n, and $2^{k(x)}$ otherwise.

For example, let $n = 13 = 1101_2$. Then
$g(4) = g(100_2) = 4 = |\{100_2, 101_2, 110_2, 111_2\}|$, but
$g(8) = g(1000_2) = 6 = |\{1000_2, 1001_2, 1010_2, 1011_2, 1100_2, 1101_2\}|$.

The number of x's with $k(x) = t$ is $\lfloor (n + 2^t)/2^{t+1} \rfloor$, since each of them is $(2z + 1)2^t$ for a non-negative integer z.

Each of these numbers contributes $2^t(n + 1 - 2^t)$ to the answer unless it is a prefix of n. All these cases can be handled and summed up in $O(\log n)$.

K. Binary Indexed Tree

Further, let $g(x)$ denote the number of y's not exceeding n such that x is a prefix of y, then $f(x) = g(x)(n+1-g(x))$.

If $k(x)$ is the number of trailing zeros in binary representation of x, then $g(x) = n - x + 1$ if x is prefix of n, and $2^{k(x)}$ otherwise.

For example, let $n = 13 = 1101_2$. Then
$g(4) = g(100_2) = 4 = |\{100_2, 101_2, 110_2, 111_2\}|$, but
$g(8) = g(1000_2) = 6 = |\{1000_2, 1001_2, 1010_2, 1011_2, 1100_2, 1101_2\}|$.

The number of x's with $k(x) = t$ is $\lfloor (n+2^t)/2^{t+1} \rfloor$, since each of them is $(2z+1)2^t$ for a non-negative integer z.

Each of these numbers contributes $2^t(n+1-2^t)$ to the answer unless it is a prefix of n. All these cases can be handled and summed up in $O(\log n)$.

Another way to deal with bit manipulations and counting is to implement some king of bitwise DP to count the same or a similar quanitities.