# Contest 3

February 18, 2019

Mikhail Tikhomirov, Artsem Zhuk

Bytedance - Moscow Workshops ICPC Programming Camp

# A. Apples

Define set $S$ as minimal by inclusion set which satisfy

1. $1 \in S$,
2. $x \in S \implies px \in S$,
3. $x \in S \implies px - 1 \in S$.

You are given $n$, find number of ways to respresent $n$ as sum of two elements of S.

# A. Apples

Note that for $x > 1$ property $x \in S$ is equivalent to two conditions:

1. $x \mod p = 0$ or $p - 1$,
2. $\lfloor x/p \rfloor \in S$.

## A. Apples

Note that for $x > 1$ property $x \in S$ is equivalent to two conditions:

1. $x \mod p = 0$ or $p - 1$,
2. $\lfloor x/p \rfloor \in S$.

So we can check if $x \in S$ in $O(\log_p x)$ time.

# A. Apples

Note that for $x > 1$ property $x \in S$ is equivalent to two conditions:

1. $x \mod p = 0$ or $p - 1$,
2. $\lfloor x/p \rfloor \in S$.

So we can check if $x \in S$ in $O(\log_p x)$ time.

Now we can write $n$ in base $p$ and go from the lowest bit. Remainder $\mod p$ must belong to set $\{-2, -1, 0, 1\}$, for each case we can count approprite number of ways.

# A. Apples

Note that for $x > 1$ property $x \in S$ is equivalent to two conditions:

1. $x \mod p = 0$ or $p - 1$,
2. $\lfloor x/p \rfloor \in S$.

So we can check if $x \in S$ in $O(\log_p x)$ time.

Now we can write $n$ in base $p$ and go from the lowest bit. Remainder $\mod p$ must belong to set $\{-2, -1, 0, 1\}$, for each case we can count approprite number of ways.

Overall complexity is $O(\log^2 n)$.

# B. Brains Eating

We have a directed graph with letters written on edges. We start at the vertex 1 and repeatedly follow a random outcoming edge, writing down letters on all edges we traverse in a string $w$. The process terminates once any of the following happens:

- $w$ contains $s$ as a substring;
- $w$ contains $t$ as a subsequence.

Find the expected number of steps until the process terminates (or that the expectation is infinite).

# B. Brains Eating

The state of matching $w$ with $s$ and $t$ during the random walk can be completely described with three numbers:

- $v$ — the current vertex of the graph;

# B. Brains Eating

The state of matching $w$ with $s$ and $t$ during the random walk can be completely described with three numbers:

- $v$ — the current vertex of the graph;
- $i$ — the length of the largest suffix of $w$ that is a prefix of $s$;

## B. Brains Eating

The state of matching $w$ with $s$ and $t$ during the random walk can be completely described with three numbers:

- $v$ — the current vertex of the graph;
- $i$ — the length of the largest suffix of $w$ that is a prefix of $s$;
- $j$ — the length of a largest common subsequence of $w$ and $t$.

# B. Brains Eating

The state of matching $w$ with $s$ and $t$ during the random walk can be completely described with three numbers:

- $v$ — the current vertex of the graph;
- $i$ — the length of the largest suffix of $w$ that is a prefix of $s$;
- $j$ — the length of a largest common subsequence of $w$ and $t$.

Construct a Markov chain with states $(v, i, j)$ and appropriate transitions.

# B. Brains Eating

The state of matching $w$ with $s$ and $t$ during the random walk can be completely described with three numbers:

- $v$ — the current vertex of the graph;
- $i$ — the length of the largest suffix of $w$ that is a prefix of $s$;
- $j$ — the length of a largest common subsequence of $w$ and $t$.

Construct a Markov chain with states $(v, i, j)$ and appropriate transitions.

The expected number of steps until termination can be found with Gaussian elimination with relations $E_{v,i,j} =$ (the expected number of steps until termination from the state $(v, i, j)$) $=$ (the average of $E_{\dots}$ for all adjacent states) $+ 1$, or $E_{v,i,j} = 0$ for terminal states.

# B. Brains Eating

The state of matching $w$ with $s$ and $t$ during the random walk can be completely described with three numbers:

- $v$ — the current vertex of the graph;
- $i$ — the length of the largest suffix of $w$ that is a prefix of $s$;
- $j$ — the length of a largest common subsequence of $w$ and $t$.

Construct a Markov chain with states $(v, i, j)$ and appropriate transitions.

The expected number of steps until termination can be found with Gaussian elimination with relations $E_{v,i,j} =$ (the expected number of steps until termination from the state $(v, i, j)$) $=$ (the average of $E_{\dots}$ for all adjacent states) $+ 1$, or $E_{v,i,j} = 0$ for terminal states.

However, $O((n|s||t|)^3)$ is too slow.

# B. Brains Eating

Note that on each step $j$ either increases by 1, or stays the same.

# B. Brains Eating

Note that on each step $j$ either increases by 1, or stays the same.

Let us compute all $E_{v,i,j}$ by decreasing of $j$. For $j < |t|$, each transition from $(v, i, j)$ follows to a $(v', i', j)$, or to a $(v', i', j + 1)$ (a state we already know $E_{v',i',j+1}$ for).

## B. Brains Eating

Note that on each step $j$ either increases by 1, or stays the same.

Let us compute all $E_{v,i,j}$ by decreasing of $j$. For $j < |t|$, each transition from $(v, i, j)$ follows to a $(v', i', j)$, or to a $(v', i', j+1)$ (a state we already know $E_{v',i',j+1}$ for).

Therefore, for each $j$ we have a $n|s| \times n|s|$ linear equation system.

# B. Brains Eating

Note that on each step $j$ either increases by 1, or stays the same.

Let us compute all $E_{v,i,j}$ by decreasing of $j$. For $j < |t|$, each transition from $(v, i, j)$ follows to a $(v', i', j)$, or to a $(v', i', j+1)$ (a state we already know $E_{v',i',j+1}$ for).

Therefore, for each $j$ we have a $n|s| \times n|s|$ linear equation system.

Complexity has now become $O(|t|(n|s|)^3)$, which (with careful implementation) should be fast enough.

## C. Classical Problem

Among $n$ points in the plane, find three non-collinear points with smallest positive triangle area.

## C. Classical Problem

If for two points $p_i, p_j$ a point $p_k$ forms a smallest (positive) area $\triangle p_i p_j p_k$,

# C. Classical Problem

If for two points $p_i, p_j$ a point $p_k$ forms a smallest (positive) area $\triangle p_i p_j p_k$,
$\Longrightarrow$ $p_k$ has the smallest (positive) distance to the line $p_i p_j$,

# C. Classical Problem

If for two points $p_i, p_j$ a point $p_k$ forms a smallest (positive) area $\triangle p_i p_j p_k$,
$\implies$ $p_k$ has the smallest (positive) distance to the line $p_i p_j$,
which is equal to $2\frac{|(p_k - p_i)\times(p_j - p_i)|}{||p_j - p_i||}$ ($\times$ is the vector cross product),

# C. Classical Problem

If for two points $p_i, p_j$ a point $p_k$ forms a smallest (positive) area $\triangle p_i p_j p_k$,
$\implies$ $p_k$ has the smallest (positive) distance to the line $p_i p_j$, which is equal to $2\frac{|(p_k - p_i) \times (p_j - p_i)|}{||p_j - p_i||}$ ($\times$ is the vector cross product),
$\implies$ $p_k$ has the value of $p_k \times (p_j - p_i)$ closest (but not equal) to $p_i \times (p_j - p_i)$.

# C. Classical Problem

For any vector $v$, sort all points by $p_i \times v$. As $v$ rotates around the origin, there are $O(n^2)$ events "points $p_i$ and $p_j$ exchange positions in the sorted order"; we process these events chronologically and maintain a correct order.

# C. Classical Problem

For any vector $v$, sort all points by $p_i \times v$. As $v$ rotates around the origin, there are $O(n^2)$ events "points $p_i$ and $p_j$ exchange positions in the sorted order"; we process these events chronologically and maintain a correct order.

When $v = p_j - p_i$ for some pair of points $p_i, p_j$, let us also look up two points non-collinear with $p_i, p_j$ and closest to $p_i$ (or $p_j$) in the current order, and update the answer.

# C. Classical Problem

For any vector $v$, sort all points by $p_i \times v$. As $v$ rotates around the origin, there are $O(n^2)$ events "points $p_i$ and $p_j$ exchange positions in the sorted order"; we process these events chronologically and maintain a correct order.

When $v = p_j - p_i$ for some pair of points $p_i, p_j$, let us also look up two points non-collinear with $p_i, p_j$ and closest to $p_i$ (or $p_j$) in the current order, and update the answer.

Note that there may be a large group of points on the line $p_i p_j$ that we have to skip, so find closest non-collinear points with binary search.

# C. Classical Problem

For any vector $v$, sort all points by $p_i \times v$. As $v$ rotates around the origin, there are $O(n^2)$ events "points $p_i$ and $p_j$ exchange positions in the sorted order"; we process these events chronologically and maintain a correct order.

When $v = p_j - p_i$ for some pair of points $p_i, p_j$, let us also look up two points non-collinear with $p_i, p_j$ and closest to $p_i$ (or $p_j$) in the current order, and update the answer.

Note that there may be a large group of points on the line $p_i p_j$ that we have to skip, so find closest non-collinear points with binary search.

$O(n^2)$ events, sort and binary search for each of them $\implies$ $O(n^2 \log n)$ complexity.

# D. Deep In The Ocean

We are given a $2k$-regular graph $G$. Find a subset of edges of $G$ such that each vertex is incident to exactly two edges in the subset.

# D. Deep In The Ocean

Let us solve the problem independently for each connected component.

Find an Eulerian tour $v_0, \ldots, v_{kn} = v_0$ in $G$, and direct all edges from $v_i$ to $v_{i+1}$ for all $i = 0, \ldots, kn - 1$. With respect to this direction, each vertex has $k$ incoming and $k$ outcoming edges.

# D. Deep In The Ocean

Let us solve the problem independently for each connected component.

Find an Eulerian tour $v_0, \ldots, v_{kn} = v_0$ in $G$, and direct all edges from $v_i$ to $v_{i+1}$ for all $i = 0, \ldots, kn - 1$. With respect to this direction, each vertex has $k$ incoming and $k$ outcoming edges.

Construct a new bipartite graph $G'$ with $2n$ vertices labelled $(i, j)$ for $i = 1, \ldots, n$ and $j = 0, 1$. For a directed edge $v \to u$ in $G$, add an edge between $(v, 0)$ and $(u, 1)$ in $G'$. This results in $G'$ being bipartite and $k$-regular.

# D. Deep In The Ocean

**Proposition**

For any $k \geqslant 1$, any bipartite $k$-regular graph contains a perfect matching.

# D. Deep In The Ocean

**Proposition**

For any $k \geqslant 1$, any bipartite $k$-regular graph contains a perfect matching.

Proof: consider any subset $X$ of the left part of the graph. It has $k|X|$ incident edges, hence it must have at least $|X|$ adjacent vertices in the right part (otherwise the total degree of adjacent vertices would be less than $k|X|$). Hall's marriage theorem now implies that a perfect matching must exist.

# D. Deep In The Ocean

Now, find a perfect matching $M$ in $G'$ with Kuhn's algorithm. After converting back to vertices of $G$ ($(v, i) \rightarrow v$), edges of $M$ are a suitable subset. This finishes the solution (and simultaneously, a proof that the answer always exists).

# D. Deep In The Ocean

Now, find a perfect matching $M$ in $G'$ with Kuhn's algorithm. After converting back to vertices of $G$ ($(v, i) \to v$), edges of $M$ are a suitable subset. This finishes the solution (and simultaneously, a proof that the answer always exists).

The hardest part is Kuhn's algorithm, hence the complexity is $O(nm)$.

# E. Endgame in Reversi

Determine a winner in a reversi position with at most 12 moves left.

# E. Endgame in Reversi

Determine a winner in a reversi position with at most 12 moves left.

Efficient brute-force with alpha-beta pruning and ~~hacks~~ optimizations.

# F. Frozen Orb

There are $n$ disjoint enemies (circles) in the plane, each enemy has a certain amount of health. We can fire a frozen orb from the origin in an arbitrary direction $\varphi$, which in turn fires $K$ packs of $M$ bolts each at particular directions w.r. to $\varphi$ as it flies. Each bolt hitting an enemy decreases its health by 1, an enemy with non-positive health dies. Choose a direction for the orb so that to kill as many enemies as possible.

# F. Frozen Orb

Let us rotate the system by $-\varphi$ degrees around the origin. The orb direction (and hence, the bolts initial locations and direction) are now fixed, and we are free to rotate all the enemies.

# F. Frozen Orb

Let us rotate the system by $-\varphi$ degrees around the origin. The orb direction (and hence, the bolts initial locations and direction) are now fixed, and we are free to rotate all the enemies.

Consider any particular enemy, and any particular bolt. What are the possible angles $\varphi$ such that the bolt hits the enemy? One can see that such $\varphi$ form at most two segments in the circle $[0, 2\pi)$; the segments can be found with casework and standard geometric primitives (such as line-circle intersection).

# F. Frozen Orb

Let us rotate the system by $-\varphi$ degrees around the origin. The orb direction (and hence, the bolts initial locations and direction) are now fixed, and we are free to rotate all the enemies.

Consider any particular enemy, and any particular bolt. What are the possible angles $\varphi$ such that the bolt hits the enemy? One can see that such $\varphi$ form at most two segments in the circle $[0, 2\pi)$; the segments can be found with casework and standard geometric primitives (such as line-circle intersection).

Now, for a particular enemy, which angles $\varphi$ result in this enemy being killed? We can generate the "hitting" segments for all $KM$ bolts, and find disjoint "killing" portions of $[0; 2\pi)$ covered by a sufficient number of segments in $O(KM \log KM)$ time.

# F. Frozen Orb

Finally, to find an angle with the largest number of enemies killed, aggregate the "killing" segments for all enemies and find a point of $[0, 2\pi)$ covered by the largest number of segments.

# F. Frozen Orb

Finally, to find an angle with the largest number of enemies killed, aggregate the "killing" segments for all enemies and find a point of $[0, 2\pi)$ covered by the largest number of segments.

The total number of segments generated is $O(NKM)$, hence the total complexity is $O(NKM \log NKM)$.

# F. Frozen Orb

Finally, to find an angle with the largest number of enemies killed, aggregate the "killing" segments for all enemies and find a point of $[0, 2\pi)$ covered by the largest number of segments.

The total number of segments generated is $O(NKM)$, hence the total complexity is $O(NKM \log NKM)$.

Discrete answer sensitive to floating-point error, so...

# F. Frozen Orb

Finally, to find an angle with the largest number of enemies killed, aggregate the "killing" segments for all enemies and find a point of $[0, 2\pi)$ covered by the largest number of segments.

The total number of segments generated is $O(NKM)$, hence the total complexity is $O(NKM \log NKM)$.

Discrete answer sensitive to floating-point error, so...

*it's time to play...*

# F. Frozen Orb

Finally, to find an angle with the largest number of enemies killed, aggregate the “killing” segments for all enemies and find a point of $[0, 2\pi)$ covered by the largest number of segments.

The total number of segments generated is $O(NKM)$, hence the total complexity is $O(NKM \log NKM)$.

Discrete answer sensitive to floating-point error, so...

*it’s time to play...*

**THE $\varepsilon$-GUESSING GAME.**

# F. Frozen Orb

Finally, to find an angle with the largest number of enemies killed, aggregate the "killing" segments for all enemies and find a point of $[0, 2\pi)$ covered by the largest number of segments.

The total number of segments generated is $O(NKM)$, hence the total complexity is $O(NKM \log NKM)$.

Discrete answer sensitive to floating-point error, so...

*it's time to play...*

**THE $\varepsilon$-GUESSING GAME.**

...or not, if test cases are friendly.

# G. Gapless Filling with Tetrominoes

Find number of filling by tetraminoes without gaps.

## G. Gapless Filling with Tetrominoes

State of dynamic programming is a mask of bottomost-leftmost 12 cells.

# G. Gapless Filling with Tetrominoes

State of dynamic programming is a mask of bottomost-leftmost 12 cells.

Number of reachable states is around 100.

# G. Gapless Filling with Tetrominoes

State of dynamic programming is a mask of bottomost-leftmost 12 cells.

Number of reachable states is around 100.

Can use matrix multiplication.

# H. Hand-made Cube

Given a sheet with eight colored flat cube nets, determine if it is possible to construct a $2 \times 2 \times 2$ cube out of them so that all inside faces are black, and each outside face has uniform, distinct color.

# H. Hand-made Cube

Parse the sheet into eight actual cubes (e.g. with DFS keeping track of cube faces and directions).

# H. Hand-made Cube

Parse the sheet into eight actual cubes (e.g. with DFS keeping track of cube faces and directions).

There are now $8! \times 3^8$ ways to construct a bicube with rotation of each particular cube.

# H. Hand-made Cube

Parse the sheet into eight actual cubes (e.g. with DFS keeping track of cube faces and directions).

There are now $8! \times 3^8$ ways to construct a bicube with rotation of each particular cube.

Can reduce to $7! \times 3^7$ options by fixing one cube in place.

# H. Hand-made Cube

Parse the sheet into eight actual cubes (e.g. with DFS keeping track of cube faces and directions).

There are now $8! \times 3^8$ ways to construct a bicube with rotation of each particular cube.

Can reduce to $7! \times 3^7$ options by fixing one cube in place.

Can reduce even more by fixing two opposite cubes, thus determining all face colors (hence all other cubes' positions).

# H. Hand-made Cube

Parse the sheet into eight actual cubes (e.g. with DFS keeping track of cube faces and directions).

There are now $8! \times 3^8$ ways to construct a bicube with rotation of each particular cube.

Can reduce to $7! \times 3^7$ options by fixing one cube in place.

Can reduce even more by fixing two opposite cubes, thus determining all face colors (hence all other cubes' positions).

Lots of technical difficulties.

# H. Hand-made Cube

Parse the sheet into eight actual cubes (e.g. with DFS keeping track of cube faces and directions).

There are now $8! \times 3^8$ ways to construct a bicube with rotation of each particular cube.

Can reduce to $7! \times 3^7$ options by fixing one cube in place.

Can reduce even more by fixing two opposite cubes, thus determining all face colors (hence all other cubes' positions).

Lots of technical difficulties.

Try not to cry. Cry a lot.

# I. Inverse LCP Problem

You are given a string. For multiple queries find lexicographically smallest pair of numbers $(i, j)$ with given $lcp(s_i, s_j) = k_q$.

# I. Inverse LCP Problem

One of solutions is build suffix tree and compute dynamic programming: for each vertex, lexicographically smallest pair of veritces with this vertex as their *lca*. Complexity is $O(n)$.

# I. Inverse LCP Problem

One of solutions is build suffix tree and compute dynamic programming: for each vertex, lexicographically smallest pair of veritces with this vertex as their *lca*. Complexity is $O(n)$.

Another one is to compute suffix array. Then *lcp* is minimum on some segment on array. Can use DSU and greedily merge segments. Complexity is $O(n \log n)$ or $O(n \log^2 n)$ depending on your laziness.

# J. Just Created Language

Interpret a program with assignment and print instructions. Instructions involve standard integer and string operators, regexp-based variable name substitution, and a few other regexp-based features (such as a shortest substring matching a regexp).

# J. Just Created Language

First, parse everything with a logic tree or recursive descent (20+ different behaviours, easy enough).

# J. Just Created Language

First, parse everything with a logic tree or recursive descent (20+ different behaviours, easy enough).

With clever implementation, interpreting can be done along with the parsing.

# J. Just Created Language

First, parse everything with a logic tree or recursive descent (20+ different behaviours, easy enough).

With clever implementation, interpreting can be done along with the parsing.

All intermediate strings are guaranteed to be short, so a lot of things can be done straightforwardly (i.e. match all variable names, match all substrings, etc.).

# J. Just Created Language

First, parse everything with a logic tree or recursive descent (20+ different behaviours, easy enough).

With clever implementation, interpreting can be done along with the parsing.

All intermediate strings are guaranteed to be short, so a lot of things can be done straightforwardly (i.e. match all variable names, match all substrings, etc.).

Matching with regexp can be done with DP ("matched a substring of a string with a sub-regexp"), or by constructing an NFA (non-deterministic finite automaton) and keeping track of reachable states for all prefixes.

# J. Just Created Language

First, parse everything with a logic tree or recursive descent (20+ different behaviours, easy enough).

With clever implementation, interpreting can be done along with the parsing.

All intermediate strings are guaranteed to be short, so a lot of things can be done straightforwardly (i.e. match all variable names, match all substrings, etc.).

Matching with regexp can be done with DP ("matched a substring of a string with a sub-regexp"), or by constructing an NFA (non-deterministic finite automaton) and keeping track of reachable states for all prefixes.

Complexity: am I a joke to you?

# J. Just Created Language

First, parse everything with a logic tree or recursive descent (20+ different behaviours, easy enough).

With clever implementation, interpreting can be done along with the parsing.

All intermediate strings are guaranteed to be short, so a lot of things can be done straightforwardly (i.e. match all variable names, match all substrings, etc.).

Matching with regexp can be done with DP ("matched a substring of a string with a sub-regexp"), or by constructing an NFA (non-deterministic finite automaton) and keeping track of reachable states for all prefixes.

Complexity: am I a joke to you?

*Seriously though, master hard implementation, it's good for you.*

# K. Killer

You have a cycle of length $2n$, and build random matching on vertices of this cycle. Find probablity that exactly $k$ edges belong to the cycle.

# K. Killer

How many matching are there on $2n$ vertices?

$$(2n-1)!! = \frac{2n!}{2^n n!}.$$

# K. Killer

How many matching are there on $2n$ vertices?

$$(2n-1)!! = \frac{2n!}{2^n n!}.$$

In how many ways we can choose $k$ disjoint pairs of neighbours on the cycle graph?

$$\binom{2n-3-(k-1)}{k-1} + \binom{2n-2-k}{k} = \binom{2n-1}{k}.$$

# K. Killer

How many matching are there on $2n$ vertices?

$$(2n-1)!! = \frac{2n!}{2^n n!}.$$

In how many ways we can choose $k$ disjoint pairs of neighbours on the cycle graph?

$$\binom{2n-3-(k-1)}{k-1} + \binom{2n-2-k}{k} = \binom{2n-1}{k}.$$

So, number of ways to chose matching and $i$ edges from cycle in this matching is

$$\binom{2n-1-i}{i}(2n-2i)!!$$

# K. Killer

Using inclusion-declusion, we get the simple formula...

## K. Killer

Using inclusion-declusion, we get the simple formula...

$$ans = \frac{\sum_{i=k}^{i=n}(-1)^{i-k}\binom{i}{k}\binom{2n-1-i}{i}(2n-2i)!!}{(2n-1)!!}$$

# K. Killer

Using inclusion-declusion, we get the simple formula...

$$ans = \frac{\sum_{i=k}^{i=n}(-1)^{i-k}\binom{i}{k}\binom{2n-1-i}{i}(2n-2i)!!}{(2n-1)!!}$$

Use BigIntegers to calculate it.

Complexity is $O(n^2)$.