Contest 2

February 17, 2019

Mikhail Tikhomirov, Artsem Zhuk

Bytedance - Moscow Workshops ICPC Programming Camp

A. Always Return A Value

Given a C++-like function with two `int` parameters and nested conditions, determine if it always/sometimes/never returns a value.

A. Always Return A Value

We can represent all possible inputs to a function as a two-dimensional plane. Let us split the plane into regions with vertical/horizontal lines by all numbers appearing in the comparisons.

A. Always Return A Value

We can represent all possible inputs to a function as a two-dimensional plane. Let us split the plane into regions with vertical/horizontal lines by all numbers appearing in the comparisons.

Let C be the number of splits; C is at most a few hundred (since the size of code is at most 1.5Kb). There are $O(C^2)$ different regions of the plane, try to substitute values from all of them and see which of them result in returning a value.

A. Always Return A Value

We can represent all possible inputs to a function as a two-dimensional plane. Let us split the plane into regions with vertical/horizontal lines by all numbers appearing in the comparisons.

Let C be the number of splits; C is at most a few hundred (since the size of code is at most 1.5Kb). There are $O(C^2)$ different regions of the plane, try to substitute values from all of them and see which of them result in returning a value.

Evaluating a function requires parsing it, e.g. with a recursive descent parser.

B. Cheese

You are given pairs (= vectors) of positive numbers (v_i, m_i).

You can *divide* **at most two** pairs , which means:

$$(v, m) \xrightarrow{\text{replace with}} (pv, pm) \text{ and } ((1-p)v, (1-p)m).$$

.

After that you want to split vectors in two groups with equal sum of both coordinates.

B. Cheese

Consider rectangles $(m_i/v_i) \times v_i$.

B. Cheese

Consider rectangles $(m_i/v_i) \times v_i$.

Total width is $V = \sum v_i$, total area is $M = \sum \frac{m_i}{v_i} \cdot v_i = \sum m_i$.

B. Cheese

Consider rectangles $(m_i/v_i) \times v_i$.

Total width is $V = \sum v_i$, total area is $M = \sum \frac{m_i}{v_i} \cdot v_i = \sum m_i$.

Sort them by width $(=v_i)$.

B. Cheese

Consider rectangles $(m_i/v_i) \times v_i$.

Total width is $V = \sum v_i$, total area is $M = \sum \frac{m_i}{v_i} \cdot v_i = \sum m_i$.

Sort them by width $(=v_i)$.

Now we want to make two cuts at distance $\frac{V}{2} = \frac{\sum v_i}{2}$, such that area between cuts is half of overall area M.

B. Cheese

Consider rectangles $(m_i/v_i) \times v_i$.

Total width is $V = \sum v_i$, total area is $M = \sum \frac{m_i}{v_i} \cdot v_i = \sum m_i$.

Sort them by width $(=v_i)$.

Now we want to make two cuts at distance $\frac{V}{2} = \frac{\sum v_i}{2}$, such that area between cuts is half of overall area M.

Use binary search!

B. Cheese

Consider rectangles $(m_i/v_i) \times v_i$.

Total width is $V = \sum v_i$, total area is $M = \sum \frac{m_i}{v_i} \cdot v_i = \sum m_i$.

Sort them by width $(=v_i)$.

Now we want to make two cuts at distance $\frac{V}{2} = \frac{\sum v_i}{2}$, such that area between cuts is half of overall area M.

Use binary search!

Complexity $O(nC)$, where C number of binary search iterations.

C. Colored Graph

You are given polygon with **all** edges and diagonals colored black and white.

Find monochromatic spanning tree without self-intersections.

C. Colored Graph

Solution exists for every coloring.

C. Colored Graph

Solution exists for every coloring.

Consider perimter of the polygon. If it is monochromatic, we can drop any edge and get the solution.

C. Colored Graph

Solution exists for every coloring.

Consider perimter of the polygon. If it is monochromatic, we can drop any edge and get the solution.

Otherwise, there is two consecutive sides of polygon of distinct colors.

C. Colored Graph

Solution exists for every coloring.

Consider perimter of the polygon. If it is monochromatic, we can drop any edge and get the solution.

Otherwise, there is two consecutive sides of polygon of distinct colors.

Remove them and solve problem recursively.

C. Colored Graph

Solution exists for every coloring.

Consider perimter of the polygon. If it is monochromatic, we can drop any edge and get the solution.

Otherwise, there is two consecutive sides of polygon of distinct colors.

Remove them and solve problem recursively.

Careful implementation (for instance with query) leads to linear number of queries of color of an edge.

C. Colored Graph

Solution exists for every coloring.

Consider perimter of the polygon. If it is monochromatic, we can drop any edge and get the solution.

Otherwise, there is two consecutive sides of polygon of distinct colors.

Remove them and solve problem recursively.

Careful implementation (for instance with query) leads to linear number of queries of color of an edge.

Can get color of any edge in $O(\log n)$ time, so overall complexity is $O(n \log n)$.

D. Cross-section

You are given non-convex polygon without self-intersections and a point.

Consider line throught the point with uniformly distributed slope.

Find expect value of length of line intersection with polygon interrior.

D. Cross-section

By linearity can solve separatly for each side.

D. Cross-section

By linearity can solve separatly for each side.

Let's write down the integral. The segment endpoints are (x_1, y_1), (x_2, y_2).

The system is

$$px_1 + (1-p)x_2 = r\cos\phi$$
$$py_1 + (1-p)y_2 = r\sin\phi$$

D. Cross-section

By linearity can solve separatly for each side.

Let's write down the integral. The segment endpoints are (x_1, y_1), (x_2, y_2).

The system is

$$\begin{aligned} px_1 + (1-p)x_2 &= r\cos\phi \\ py_1 + (1-p)y_2 &= r\sin\phi \end{aligned}$$

After solving, we get that

$$r = \frac{C}{A\cos\phi + B\sin\phi} = \frac{D}{\cos(\phi + \phi_0)}$$

D. Cross-section

Integral

$$\int \frac{d\phi}{\cos\phi}$$

D. Cross-section

Integral

$$\int \frac{d\phi}{\cos\phi}$$

is left as an excersice for the reader.

D. Cross-section

Integral

$$\int \frac{d\phi}{\cos\phi}$$

is left as an excersice for the reader.

Overall complexity is $O(n)$.

E. Grouping by prefixes

Given a set of string $s_1, \ldots, s_n$, count the number of sets of non-empty strings $t_1, \ldots, t_m$ such that each string s_i contains exactly one t_j as a prefix.

E. Grouping by prefixes

Note that if s_i contains s_j as a prefix, we can get rid of s_i without changing the answer.

E. Grouping by prefixes

Note that if s_i contains s_j as a prefix, we can get rid of s_i without changing the answer.

Construct a trie with strings $s_1, \ldots, s_n$; strings $s_1, \ldots, s_n$ are exactly the leaves of this trie.

E. Grouping by prefixes

Note that if s_i contains s_j as a prefix, we can get rid of s_i without changing the answer.

Construct a trie with strings $s_1, \ldots, s_n$; strings $s_1, \ldots, s_n$ are exactly the leaves of this trie.

An *antichain* in the trie is a set of vertices with no vertex being an ancestor of any other. We have to count the number of inclusion-maximal antichains of size m in the trie.

E. Grouping by prefixes

Consider any maximal antichain C, and the subtree T_v of any vertex v. Observe that either T_v contains no vertices of C (and is covered somewhere above), or the restriction of C to T_v is a maximal antichain in T_v.

E. Grouping by prefixes

Consider any maximal antichain C, and the subtree T_v of any vertex v. Observe that either T_v contains no vertices of C (and is covered somewhere above), or the restriction of C to T_v is a maximal antichain in T_v.

We can now use dynamic programming $dp_{v,s}$ — the number of maximal antichains of size s in T_v. The transitions are:

- choose maximal antichains in subtrees of all children of v (if any);

E. Grouping by prefixes

Consider any maximal antichain C, and the subtree T_v of any vertex v. Observe that either T_v contains no vertices of C (and is covered somewhere above), or the restriction of C to T_v is a maximal antichain in T_v.

We can now use dynamic programming $dp_{v,s}$ — the number of maximal antichains of size s in T_v. The transitions are:

- choose maximal antichains in subtrees of all children of v (if any);
- take v as the only member of the antichain.

E. Grouping by prefixes

Consider any maximal antichain C, and the subtree T_v of any vertex v. Observe that either T_v contains no vertices of C (and is covered somewhere above), or the restriction of C to T_v is a maximal antichain in T_v.

We can now use dynamic programming $dp_{v,s}$ — the number of maximal antichains of size s in T_v. The transitions are:

- choose maximal antichains in subtrees of all children of v (if any);
- take v as the only member of the antichain.

Process multiple children of v with a sub-DP $dp'_{v,i,s}$ — the answer after processing i children of v.

E. Grouping by prefixes

To reduce the trie size, compress edge chains with no branches; multiply the answer by the chain length each time you pick a vertex in this chain. After compression, the trie will only contain $O(n)$ vertices.

E. Grouping by prefixes

To reduce the trie size, compress edge chains with no branches; multiply the answer by the chain length each time you pick a vertex in this chain. After compression, the trie will only contain $O(n)$ vertices.

Another (fairly standard) optimization is to only consider values of $dp_{v,s}$ with s not exceeding the size of T_v. With this optimization, one can show that the total complexity of this solution is $O(n^2)$.

F. Inserting Lines

We have an infinite table filled with values $A_{x,y} = (x, y)$. Process three types of queries (online):

- insert a row: for a parameter y_i shift all cells in the table with $y \geqslant y_i$ up one cell, assign $A_{x,y_i} = (x, i)$, where i is the query index;
- insert a column (similarly);
- return the value of a given cell (x_i, y_i).

F. Inserting Lines

Let us maintain sets X and Y of resp. columns and rows that contain cells not originally present in the table; for each element of X and Y also store the query index when resp. row/column was inserted.

F. Inserting Lines

Let us maintain sets X and Y of resp. columns and rows that contain cells not originally present in the table; for each element of X and Y also store the query index when resp. row/column was inserted.

Inserting a row/column results in inserting a new value into X/Y and shifting a suffix by 1. We can implement this e.g. with two treaps with lazy addition.

F. Inserting Lines

Let us maintain sets X and Y of resp. columns and rows that contain cells not originally present in the table; for each element of X and Y also store the query index when resp. row/column was inserted.

Inserting a row/column results in inserting a new value into X/Y and shifting a suffix by 1. We can implement this e.g. with two treaps with lazy addition.

How to answer the value query (x, y)? If $x \notin X$ and $y \notin Y$, then the x-coordinate of the answer is $x_{ans} = x - \delta_x$, where δ_x is the number of elements of X less than x (i.e. the number of shifts that happened to the original cell now in position (x, y)). y_{ans} can be determined in the same way.

F. Inserting Lines

If $x \in X$ and $y \notin Y$, then $x_{ans} = id_x$, where id_x is the query index of inserting a column now at position x. The $y \in Y, x \notin X$ case is treated similarly.

F. Inserting Lines

If $x \in X$ and $y \notin Y$, then $x_{ans} = id_x$, where id_x is the query index of inserting a column now at position x. The $y \in Y, x \notin X$ case is treated similarly.

If $x \in X$ and $y \in Y$, then the later of the two insertions applies according to the above.

F. Inserting Lines

If $x \in X$ and $y \notin Y$, then $x_{ans} = id_x$, where id_x is the query index of inserting a column now at position x. The $y \in Y, x \notin X$ case is treated similarly.

If $x \in X$ and $y \in Y$, then the later of the two insertions applies according to the above.

Now to find the remaining coordinate; WLOG $x_{ans} = id_x$ and we want to find y_{ans}.

F. Inserting Lines

If $x \in X$ and $y \notin Y$, then $x_{ans} = id_x$, where id_x is the query index of inserting a column now at position x. The $y \in Y, x \notin X$ case is treated similarly.

If $x \in X$ and $y \in Y$, then the later of the two insertions applies according to the above.

Now to find the remaining coordinate; WLOG $x_{ans} = id_x$ and we want to find y_{ans}.

Similar to the very first case, $y_{ans} = y - \delta_y$, with δ_y being the number of shifts happened to the cell (id_x, y_{ans}) **after** the column id_x was inserted.

F. Inserting Lines

We can find δ_y as $c_{now} - c_{id_x}$, where c_{now} is the number of elements of Y less then y, and c_{id_x} is the number of elements of Y at the moment id_x that are **now** at positions less than y.

F. Inserting Lines

We can find δ_y as $c_{now} - c_{id_x}$, where c_{now} is the number of elements of Y less then y, and c_{id_x} is the number of elements of Y at the moment id_x that are **now** at positions less than y.

To compute this, make Y (and X) persistent. Let us perform a descent in the version id_x of Y (similar to how we compute the number of element less than y).

F. Inserting Lines

We can find δ_y as $c_{now} - c_{id_x}$, where c_{now} is the number of elements of Y less then y, and c_{id_x} is the number of elements of Y at the moment id_x that are **now** at positions less than y.

To compute this, make Y (and X) persistent. Let us perform a descent in the version id_x of Y (similar to how we compute the number of element less than y).

Each time we decide to go left or right of the current vertex, we look up the current coordinate of the resp. column in the current version of Y. Note that to do this we need to traverse to the root of Y and aggregate the delayed additions.

F. Inserting Lines

We can find δ_y as $c_{now} - c_{id_x}$, where c_{now} is the number of elements of Y less then y, and c_{id_x} is the number of elements of Y at the moment id_x that are **now** at positions less than y.

To compute this, make Y (and X) persistent. Let us perform a descent in the version id_x of Y (similar to how we compute the number of element less than y).

Each time we decide to go left or right of the current vertex, we look up the current coordinate of the resp. column in the current version of Y. Note that to do this we need to traverse to the root of Y and aggregate the delayed additions.

This makes for an $O(n \log^2 n)$ solution.

G. Paths in Tree

You are given a tree. Select at most k non-intersecting paths with maximal total length.

G. Paths in Tree

Dynamic programming `d[v][k][b]` — maximal length of k paths in subtree with root v, b — boolean with meaning if we have path's end in v.

G. Paths in Tree

Dynamic programming `d[v][k][b]` — maximal length of k paths in subtree with root v, b — boolean with meaning if we have path's end in v.

Merge them naively:

$$\texttt{d\_new[v][k][b]} = \sum_i \texttt{d[v][i][b} \pm \texttt{1]} \cdot \texttt{d[u][k-i} \pm \texttt{1][b} \pm \texttt{1]}$$

G. Paths in Tree

Dynamic programming `d[v][k][b]` — maximal length of k paths in subtree with root v, b — boolean with meaning if we have path's end in v.

Merge them naively:

$$\texttt{d\_new[v][k][b]} = \sum_i \texttt{d[v][i][b} \pm \texttt{1]} \cdot \texttt{d[u][k-i} \pm \texttt{1][b} \pm \texttt{1]}$$

with optimization that k and i go only up to size of a subtree.

G. Paths in Tree

Trivial complexity bound is $O(nk^2)$, but let's consider it more carefully.

G. Paths in Tree

Trivial complexity bound is $O(nk^2)$, but let's consider it more carefully.

Consider deepest subtrees with size $\geqslant k$. We have at most n/k of them, in each of them we use $O(k^2)$ operations — $O(nk)$ overall.

G. Paths in Tree

Trivial complexity bound is $O(nk^2)$, but let's consider it more carefully.

Consider deepest subtrees with size $\geqslant k$. We have at most n/k of them, in each of them we use $O(k^2)$ operations — $O(nk)$ overall.

Cut these subtrees, remaining tree has at most n/k leaves. That means it has at most $O(n/k)$ internal vertices with more than one child. Only in this vertices we merge in $O(k^2)$. Again, overall complexity is $O(nk)$.

H. Restoring Points

Restore an arrangement of $\geqslant 11$ integer points by a multiset of squared pairwise distances.

H. Restoring Points

One has to implement an efficient brute-force. Some ideas:

- We can generate all possible vectors (Δ_x, Δ_y) such $\Delta_x^2 + \Delta_y^2$ is in the given multiset.

H. Restoring Points

One has to implement an efficient brute-force. Some ideas:

- We can generate all possible vectors (Δ_x, Δ_y) such $\Delta_x^2 + \Delta_y^2$ is in the given multiset.
- Start with $p_0 = (0, 0)$, add points one by one, try all options for the vector $p_i - p_0$, check if distance multiplicities are okay.

H. Restoring Points

One has to implement an efficient brute-force. Some ideas:

- We can generate all possible vectors (Δ_x, Δ_y) such $\Delta_x^2 + \Delta_y^2$ is in the given multiset.
- Start with $p_0 = (0, 0)$, add points one by one, try all options for the vector $p_i - p_0$, check if distance multiplicities are okay.
- Trying short vectors first helps.

H. Restoring Points

One has to implement an efficient brute-force. Some ideas:

- We can generate all possible vectors (Δ_x, Δ_y) such $\Delta_x^2 + \Delta_y^2$ is in the given multiset.
- Start with $p_0 = (0, 0)$, add points one by one, try all options for the vector $p_i - p_0$, check if distance multiplicities are okay.
- Trying short vectors first helps.
- Random shuffle helps.

H. Restoring Points

One has to implement an efficient brute-force. Some ideas:

- We can generate all possible vectors (Δ_x, Δ_y) such $\Delta_x^2 + \Delta_y^2$ is in the given multiset.
- Start with $p_0 = (0, 0)$, add points one by one, try all options for the vector $p_i - p_0$, check if distance multiplicities are okay.
- Trying short vectors first helps.
- Random shuffle helps.
- Non-asymptotic optimizations are crucial (pruning already used distances, $O(1)$ whenever possible, etc).

I. Security Policy

We have several non-intersecting countries (= convex polygons) on a plane. Each country has a standing on five different topics. Choose one of five topics for each country so that no two neighbouring countries are assigned the same topic they share a standing on.

I. Security Policy

In the worst case all countries have the same standing on each topic, then we have to color the planar graph of countries in five colors. Since this solves the problem for any instance, we will just do that.

I. Security Policy

In the worst case all countries have the same standing on each topic, then we have to color the planar graph of countries in five colors. Since this solves the problem for any instance, we will just do that.

Observation: any planar graph has a vertex v of degree $\leqslant 5$. Let's remove this vertex from the graph and color the rest recursively.

I. Security Policy

Add v back to the graph. If not all five colors are present among its neighbours, choose a free one.

I. Security Policy

Add v back to the graph. If not all five colors are present among its neighbours, choose a free one.

Otherwise, let's number the neighbours and colors 1 through 5 in cyclic order around the vertex. Consider a connected component of colors 1 and 3 containing the neighbour 1. If it does not contain the neighbour 3, swap colors within the component and we are now free to use color 1.

I. Security Policy

Add v back to the graph. If not all five colors are present among its neighbours, choose a free one.

Otherwise, let's number the neighbours and colors 1 through 5 in cyclic order around the vertex. Consider a connected component of colors 1 and 3 containing the neighbour 1. If it does not contain the neighbour 3, swap colors within the component and we are now free to use color 1.

In the other case, the neighbour 2 is isolated from 4 by a $1-3$ path, hence we can swap its $2-4$ component and use color 2.

I. Security Policy

Add v back to the graph. If not all five colors are present among its neighbours, choose a free one.

Otherwise, let's number the neighbours and colors 1 through 5 in cyclic order around the vertex. Consider a connected component of colors 1 and 3 containing the neighbour 1. If it does not contain the neighbour 3, swap colors within the component and we are now free to use color 1.

In the other case, the neighbour 2 is isolated from 4 by a $1-3$ path, hence we can swap its $2-4$ component and use color 2.

This can be implemented in $O(n^2)$ time (probably even faster, but there's no need to do that).

J. Encryption

You are given $n = \frac{m\phi(m)}{2}$, find m.

J. Encryption

Factorize n using Pollard's rho algorithm. Now we can greedily take prime divisors of n, starting from the biggest. Complexity — $O(n^{.25})$.

J. Encryption

Alternative solution.

Divide n by all prime divisors less than $C \cdot n^{.25}$. Define their product by X, remaining product as Y.

- If $Y = 1$, we have the factorization.
- Otherwise, $Y = P(P-1)/D$, where D is some divisor of X.
  - Case $X < Y$. Then $X < n^{.5}$ and we can iterate over divisors of X and solve quadratic equation.
  - Otherwise, we can factorize Y in $O(\sqrt{Y}) = O(n^{.25})$ time.

Be careful with case $n = 1$.

K. Triangle and Segment

You are given points on a plane, find number of ways to draw a triangle and a segment with vertices in given set without intersections.

K. Triangle and Segment

Calculate number of **intersecting** triangle and segment.

K. Triangle and Segment

Calculate number of **intersecting** triangle and segment.

For each segment and ray can calculate number of points underneath it. Now we can calculate number of points in any triangle or infinite angle in $O(1)$.

K. Triangle and Segment

Calculate number of **intersecting** triangle and segment.

For each segment and ray can calculate number of points underneath it. Now we can calculate number of points in any triangle or infinite angle in $O(1)$.

In case when one endpoint inside the triangly, we can iterate through all traingles and count number of points inside in $O(1)$ — $O(n^3)$ overall.

K. Triangle and Segment

Calculate number of **intersecting** triangle and segment.

For each segment and ray can calculate number of points underneath it. Now we can calculate number of points in any triangle or infinite angle in $O(1)$.

In case when one endpoint inside the triangly, we can iterate through all traingles and count number of points inside in $O(1)$ — $O(n^3)$ overall.

If both endpoins lie outside the triangle, we can iterate through segments and cutted out vertex of the triangle. After that, we can calcualte number of points inside angle in $O(1)$, and substract number of points inside the triangle. Overall $O(n^3)$.

K. Triangle and Segment

Calculate number of **intersecting** triangle and segment.

For each segment and ray can calculate number of points underneath it. Now we can calculate number of points in any triangle or infinite angle in $O(1)$.

In case when one endpoint inside the triangly, we can iterate through all traingles and count number of points inside in $O(1)$ — $O(n^3)$ overall.

If both endpoins lie outside the triangle, we can iterate through segments and cutted out vertex of the triangle. After that, we can calcualte number of points inside angle in $O(1)$, and substract number of points inside the triangle. Overall $O(n^3)$.

There are alternative solutions in $O(n^3 \log n)$ and $O(n^4/32)$.