Training Contest 4 Editorial

May 3, 2021

Mikhail Tikhomirov, Filipp Rukhovich & Gleb Evstropov

Bytedance-Moscow Workshops Training Camp, 2021

A. A Pyramidal Task

Given a pyramid filled with integers and many sub-pyramids as queries, find maximum integer in each subpyramid.

A. A Pyramidal Task

The idea is very close to the idea of Sparse Table data structure.

A. A Pyramidal Task

The idea is very close to the idea of Sparse Table data structure.

After generating all integers of the pyramid, calculate an array $d[i][j][l]$ for any $1 \leqslant j \leqslant i \leqslant n, 0 \leqslant l, i + 2^l - 1 \leqslant n$, such that $d[i][j][l]$ is maximum integers in sub-pyramid with top in (i, j) and side length 2^l.

A. A Pyramidal Task

The idea is very close to the idea of Sparse Table data structure.

After generating all integers of the pyramid, calculate an array $d[i][j][l]$ for any $1 \leqslant j \leqslant i \leqslant n, 0 \leqslant l, i + 2^l - 1 \leqslant n$, such that $d[i][j][l]$ is maximum integers in sub-pyramid with top in (i, j) and side length 2^l.

Each $d[i][j][l]$ can be computed in $O(1)$ using the following formula

$d[i][j][l+2] = max(d[i][j][l+1], d[i+2^l][j][l+1], d[i+2^l][j+2^l][l+1], d[i+2^{l+1}][j][l], d[i+2^{l+1}][j+2^l][l], d[i+2^{l+1})][j+2^{l+1}][l])$:

A. A Pyramidal Task

The array d can be calculated in $O(n^2 logn)$ time.

A. A Pyramidal Task

The array d can be calculated in $O(n^2 logn)$ time.

We need to show that having d, we can find answer for each query (x, y, len) in $O(1)$ time.

A. A Pyramidal Task

The array d can be calculated in $O(n^2 logn)$ time.

We need to show that having d, we can find answer for each query (x, y, len) in $O(1)$ time.

Let l be maximal integer such that $2^l \leqslant len$.

A. A Pyramidal Task

The array d can be calculated in $O(n^2 logn)$ time.

We need to show that having d, we can find answer for each query (x, y, len) in $O(1)$ time.

Let l be maximal integer such that $2^l \leqslant len$.

We can cover sub-pyramid (x, y, len) by some number of sub-pyramids of side length 2^l; in fact, this number is 6 if $2^l \cdot 1.5 < len$, $len > 3$; otherwise, this number is 3. Exact coordinates of these sub-pyramids to used can be computer from picture.

A. A Pyramidal Task

The array d can be calculated in $O(n^2 logn)$ time.

We need to show that having d, we can find answer for each query (x, y, len) in $O(1)$ time.

Let l be maximal integer such that $2^l \leqslant len$.

We can cover sub-pyramid (x, y, len) by some number of sub-pyramids of side length 2^l; in fact, this number is 6 if $2^l \cdot 1.5 < len, len > 3$; otherwise, this number is 3. Exact coordinates of these sub-pyramids to used can be computer from picture.

Having these 6 (or less) pyramids of length 2^l and d, we can find answer in $O(1)$ time, QED.

B. Big Storm

There are n bees on the plane; in one second, no more than on bee can move in any allowed directions. The set of allowed directions is a subset of eight basic directions. The problem is to find a minimum time bees need to meet alltogether in one point.

B. Big Storm

To reduce the number of cases to deal with we can split the set of integer points into four subsets by parity of each of the coordinates, i.e. $\{(2 \cdot z_1 + a, 2 \cdot z_2 + b) | z_1, z_2 \in \mathbb{Z}\}$; here $a, b \in \{0, 1\}$.

B. Big Storm

To reduce the number of cases to deal with we can split the set of integer points into four subsets by parity of each of the coordinates, i.e. $\{(2 \cdot z_1 + a, 2 \cdot z_2 + b) | z_1, z_2 \in \mathbb{Z}\}$; here $a, b \in \{0, 1\}$.

If we find the answer for each of these four subsets we only have to pick the optimal one at the end. Now we consider some a and b are fixed.

B. Big Storm

To reduce the number of cases to deal with we can split the set of integer points into four subsets by parity of each of the coordinates, i.e. $\{(2 \cdot z_1 + a, 2 \cdot z_2 + b) | z_1, z_2 \in \mathbb{Z}\}$; here $a, b \in \{0, 1\}$.

If we find the answer for each of these four subsets we only have to pick the optimal one at the end. Now we consider some a and b are fixed.

Let $f_i(z_1, z_2)$ be the minimum number of moves the i-th bee needs to make in order to achieve the point $(2 \cdot z_1 + a, 2 \cdot z_2 + b)$; if the i-th bee is unable to reach this point, value of $f_i(z_1, z_2)$ is undefined.

B. Big Storm

Our goal is to minimize $f(z_1, z_2) := \sum_{i=1}^{n} f_i(z_1, z_2)$.

B. Big Storm

Our goal is to minimize $f(z_1, z_2) := \sum_{i=1}^{n} f_i(z_1, z_2)$.

It can be proved that $f_i(z_1, z_2)$ is convex downward on it's domain; then, $f_(z_1, z_2)$ is convex downward on it's domain as sum of convex downward functions.

B. Big Storm

Our goal is to minimize $f(z_1, z_2) := \sum_{i=1}^{n} f_i(z_1, z_2)$.

It can be proved that $f_i(z_1, z_2)$ is convex downward on it's domain; then, $f_(z_1, z_2)$ is convex downward on it's domain as sum of convex downward functions.

The above means that if every $f_i(z_1, z_2)$ can be computed for any (z_1, z_2) in $O(1)$ time, the problem can be solved in $O(n \log^2 C)$ time using nested ternary search. Here C refers to the upper bound of the answer position (for example, 10^7).

B. Big Storm

Our goal is to minimize $f(z_1, z_2) := \sum_{i=1}^{n} f_i(z_1, z_2)$.

It can be proved that $f_i(z_1, z_2)$ is convex downward on it's domain; then, $f_(z_1, z_2)$ is convex downward on it's domain as sum of convex downward functions.

The above means that if every $f_i(z_1, z_2)$ can be computed for any (z_1, z_2) in $O(1)$ time, the problem can be solved in $O(n \log^2 C)$ time using nested ternary search. Here C refers to the upper bound of the answer position (for example, 10^7).

However, computing $f_i(z_1, z_2)$ turns out to be tricky.

B. Big Storm

First, we notices that it's enough to be able to calculate $f_i(z_1, z_2)$ as if we want to achieve point (x, y) from point $(0, 0)$;

B. Big Storm

First, we notices that it's enough to be able to calculate $f_i(z_1, z_2)$ as if we want to achieve point (x, y) from point $(0, 0)$;

Second, one can prove that for any (x, y) achievable from $(0, 0)$, there is an optimal solution that uses no more than three different types of steps; moreover, one of these types is used no more than once!

B. Big Storm

First, we notices that it's enough to be able to calculate $f_i(z_1, z_2)$ as if we want to achieve point (x, y) from point $(0, 0)$;

Second, one can prove that for any (x, y) achievable from $(0, 0)$, there is an optimal solution that uses no more than three different types of steps; moreover, one of these types is used no more than once!

Formally, there exists a triple of plane vectors (v_1, v_2, v_3) from set of allowed steps described in input (here, we assume that, for example NW is vector $(1, -1)$, S is $(-1, 0)$ etc.) and non-negative integers c_1, c_2 and c_3, such that $c_1 \leqslant 1$ and $(x, y) = c_1 \cdot v_1 + c_2 \cdot v_2 + c_3 \cdot v_3$; length of the path is $c_1 + c_2 + c_3$.

B. Big Storm

First, we notices that it's enough to be able to calculate $f_i(z_1, z_2)$ as if we want to achieve point (x, y) from point $(0, 0)$;

Second, one can prove that for any (x, y) achievable from $(0, 0)$, there is an optimal solution that uses no more than three different types of steps; moreover, one of these types is used no more than once!

Formally, there exists a triple of plane vectors (v_1, v_2, v_3) from set of allowed steps described in input (here, we assume that, for example NW is vector $(1, -1)$, S is $(-1, 0)$ etc.) and non-negative integers c_1, c_2 and c_3, such that $c_1 \leqslant 1$ and $(x, y) = c_1 \cdot v_1 + c_2 \cdot v_2 + c_3 \cdot v_3$; length of the path is $c_1 + c_2 + c_3$.

Using the above fact, we try all possibilies for c_1, v_1, v_2 and v_3 (no more than $9 \cdot 8 \cdot 7/2 = 252$); for each variant, we should solve the equation $c_2 \cdot v_2 + c_3 \cdot v_3 = v$, $v := (x, y) - c_1 \cdot v_1$.

if v_2 and v_3 are collinear, then it's enough to suppose that $c_2 = 0$ or $c_3 = 0$ and solve the equation of type $c \cdot u = v$ for some given plane vectors u and v (if u and v are collinear, then $c = (u, v)/(u, u)$ where (u, v) stands for the dot product of vectors u and v.

if v_2 and v_3 are collinear, then it's enough to suppose that $c_2 = 0$ or $c_3 = 0$ and solve the equation of type $c \cdot u = v$ for some given plane vectors u and v (if u and v are collinear, then $c = (u, v)/(u, u)$ where (u, v) stands for the dot product of vectors u and v.

Otherwise, we know that $[v, v_2] = [c_2 \cdot v_2 + c_3 \cdot v_3, v_2] = c_3 \cdot [v_3, v_2]$; so, $c_3 = [v, v_2]/[v_3, v_2]$; similarly, $v_3 = [v, v_3]/[v_2, v_3]$.

if v_2 and v_3 are collinear, then it's enough to suppose that $c_2 = 0$ or $c_3 = 0$ and solve the equation of type $c \cdot u = v$ for some given plane vectors u and v (if u and v are collinear, then $c = (u, v)/(u, u)$ where (u, v) stands for the dot product of vectors u and v.

Otherwise, we know that $[v, v_2] = [c_2 \cdot v_2 + c_3 \cdot v_3, v_2] = c_3 \cdot [v_3, v_2]$; so, $c_3 = [v, v_2]/[v_3, v_2]$; similarly, $v_3 = [v, v_3]/[v_2, v_3]$.

For each way to select c_1, v_1, v_2 and v_3 we can find the answer (if exist) in $O(1)$ time; thus, we can compute f_i in $O(1)$ and solve the problem in $O(n \log^2 C)$ time.

C. Cheese!

Given lists of A-type and B-type cuts of flat rectangular piece of cheese of width w and height h, and positions of n holes, we are to find a maximum possible number of holes in one slice.

C. Cheese!

As no cuts of the same type intersect, their order remains the same along any vertical (or horizontal, whatever the type is) line passing through the piece of cheese. Sort all A-type cuts by y_l and B-type cuts by x_b.

C. Cheese!

As no cuts of the same type intersect, their order remains the same along any vertical (or horizontal, whatever the type is) line passing through the piece of cheese. Sort all A-type cuts by y_l and B-type cuts by x_b.

Then, each slice can be encoded by two numbers a and b, where a stands for the A-type cut providing left border of this slice and b stands for the B-type cut that is bottom border of this slice.

C. Cheese!

As no cuts of the same type intersect, their order remains the same along any vertical (or horizontal, whatever the type is) line passing through the piece of cheese. Sort all A-type cuts by y_l and B-type cuts by x_b.

Then, each slice can be encoded by two numbers a and b, where a stands for the A-type cut providing left border of this slice and b stands for the B-type cut that is bottom border of this slice.

For each point $p_i = (x_i, y_i)$, we can find corresponding slice (a_i, b_i) $O(\log n)$ time using binary search twice.

C. Cheese!

As no cuts of the same type intersect, their order remains the same along any vertical (or horizontal, whatever the type is) line passing through the piece of cheese. Sort all A-type cuts by y_l and B-type cuts by x_b.

Then, each slice can be encoded by two numbers a and b, where a stands for the A-type cut providing left border of this slice and b stands for the B-type cut that is bottom border of this slice.

For each point $p_i = (x_i, y_i)$, we can find corresponding slice (a_i, b_i) $O(\log n)$ time using binary search twice.

The answer is maximum number of equal pairs (a_i, b_i) in list of pairs $(a_1, b_1), \ldots, (a_n, b_n)$. There are many ways to compute this in $O(n \log n)$ or $O(n)$ time. For examlpe, *std::map¡pair¡int, int¿, int¿* can be used in C++.

D. Direct The Streets

Choose directions for edges of a planar graph so that outdegree of each vertex is at most 3.

D. Direct The Streets

Proposition

In a planar graph with $n \geqslant 3$ where are at most $3n - 6$ edges.

D. Direct The Streets

Proposition

In a planar graph with $n \geqslant 3$ where are at most $3n - 6$ edges.

Proof: write Euler's formula $E - V + F = 2$, optimize for E.

D. Direct The Streets

Proposition

In a planar graph with $n \geqslant 3$ where are at most $3n - 6$ edges.

Proof: write Euler's formula $E - V + F = 2$, optimize for E.

Claim

Suppose that some directions are chosen for all edges. If there is a vertex v with outdegree > 3, then we can find a directed path from v to u, where u has outdegree $\leqslant 2$.

D. Direct The Streets

Proposition

In a planar graph with $n \geqslant 3$ where are at most $3n - 6$ edges.

Proof: write Euler's formula $E - V + F = 2$, optimize for E.

Claim

Suppose that some directions are chosen for all edges. If there is a vertex v with outdegree > 3, then we can find a directed path from v to u, where u has outdegree $\leqslant 2$.

Proof: assume the contrary. Consider the subgraph (V, E) induced by all vertices reachable from v (we add an edge in the subgraph if both its endpoints are reachable). Each vertex in the subgraph has outdegree $\geqslant 3$, hence we have $|E| \geqslant 3|V|$, in violation of the former proposition.

D. Direct The Streets

It follows that we can repeatedly "fix" vertices with large outdegree by finding $v \to u$ paths and reversing the paths (note that only outdegrees of v and u are changed).

D. Direct The Streets

It follows that we can repeatedly “fix” vertices with large outdegree by finding $v \to u$ paths and reversing the paths (note that only outdegrees of v and u are changed).

Let us process v one by one and fix everything by running *BFS* and explicitly reversing paths.

D. Direct The Streets

It follows that we can repeatedly "fix" vertices with large outdegree by finding $v \to u$ paths and reversing the paths (note that only outdegrees of v and u are changed).

Let us process v one by one and fix everything by running *BFS* and explicitly reversing paths.

This is basically finding maximal flow with Dinic's algorithm. Since all edges have unit capacities, the complexity is $O(n\sqrt{n})$, since $m = O(n)$.

E. Enormous Numbers

You are given a closed non-self-intersecting polyline such that each its segments is parallel to Ox or Oy. Compute the sum of all integers written in cells inside the polyline modulo $10^9 + 7$, if the cell (p, q) has number $p! \cdot q! mod 10^9 + 7$.

E. Enormous Numbers

First idea is to find the answer by the same way as we're calculating area of polygon by method of trapeziods.

E. Enormous Numbers

First idea is to find the answer by the same way as we're calculating area of polygon by method of trapeziods.

Suppose that vertices of polyline are given in clockwise order (if not, we can reverse the polyline); let *ans* be a variable which contains the current sum (equal to the answer at the end). Initially, $ans := 0$.

E. Enormous Numbers

First idea is to find the answer by the same way as we're calculating area of polygon by method of trapeziods.

Suppose that vertices of polyline are given in clockwise order (if not, we can reverse the polyline); let ans be a variable which contains the current sum (equal to the answer at the end). Initially, $ans := 0$.

Consider sides of the polyline one by one. Let $e = ((x_1, y_1), (x_2, y_2))$ be the current edge.

E. Enormous Numbers

First idea is to find the answer by the same way as we're calculating area of polygon by method of trapeziods.

Suppose that vertices of polyline are given in clockwise order (if not, we can reverse the polyline); let ans be a variable which contains the current sum (equal to the answer at the end). Initially, $ans := 0$.

Consider sides of the polyline one by one. Let $e = ((x_1, y_1), (x_2, y_2))$ be the current edge.

If e is vertical ($x_1 = x_2$), then we do not change ans.

E. Enormous Numbers

First idea is to find the answer by the same way as we're calculating area of polygon by method of trapeziods.

Suppose that vertices of polyline are given in clockwise order (if not, we can reverse the polyline); let ans be a variable which contains the current sum (equal to the answer at the end). Initially, $ans := 0$.

Consider sides of the polyline one by one. Let $e = ((x_1, y_1), (x_2, y_2))$ be the current edge.

If e is vertical ($x_1 = x_2$), then we do not change ans.

if $y_1 = y_2$ and $x_1 < x_2$, we increase ans by value $\sum\limits_{x=x_1}^{x_2-1} \sum\limits_{y=0}^{y_1-1} (x! * y!)$.

E. Enormous Numbers

First idea is to find the answer by the same way as we're calculating area of polygon by method of trapeziods.

Suppose that vertices of polyline are given in clockwise order (if not, we can reverse the polyline); let *ans* be a variable which contains the current sum (equal to the answer at the end). Initially, $ans := 0$.

Consider sides of the polyline one by one. Let $e = ((x_1, y_1), (x_2, y_2))$ be the current edge.

If e is vertical ($x_1 = x_2$), then we do not change *ans*.

if $y_1 = y_2$ and $x_1 < x_2$, we increase *ans* by value $\sum\limits_{x=x_1}^{x_2-1} \sum\limits_{y=0}^{y_1-1} (x! * y!)$.

if $y_1 = y_2$ and $x_1 > x_2$, we subtract $\sum\limits^{x_1-1} \sum\limits^{y_1-1} (x! \cdot y!)$ from *ans*.

To complete the solution, we should be able to calculate function

$$f(x_1, x_2, y_1) := \sum_{x=x_1}^{x_2-1} \sum_{y=0}^{y_1-1} (x! \cdot y!).$$

To complete the solution, we should be able to calculate function
$$f(x_1, x_2, y_1) := \sum_{x=x_1}^{x_2-1} \sum_{y=0}^{y_1-1} (x! \cdot y!).$$

Note that
$$f(x_1, x_2, y_1) = (\sum_{x=x_1}^{x_2-1} (x!) \cdot \sum_{y=0}^{y_1-1} (y!) = (g(x_2) - g(x_1)) \cdot g(y_1) \text{ for}$$
$$g(z_0) := \sum_{z=0}^{z_0-1} (z!), 0 \leqslant z \leqslant 10^9.$$

To complete the solution, we should be able to calculate function $f(x_1, x_2, y_1) := \sum_{x=x_1}^{x_2-1} \sum_{y=0}^{y_1-1} (x! \cdot y!)$.

Note that
$f(x_1, x_2, y_1) = (\sum_{x=x_1}^{x_2-1} (x!) \cdot \sum_{y=0}^{y_1-1} (y!) = (g(x_2) - g(x_1)) \cdot g(y_1)$ for
$g(z_0) := \sum_{z=0}^{z_0-1} (z!), 0 \leqslant z \leqslant 10^9$.

Let C be equal to 1 000 000. Precalc and store in the program code two arrays: $fc[i] := (i\ddot{C})!$ and $gc[i] := g(i \cdot C), 0 \leqslant i \leqslant \lfloor \frac{10^9}{C} \rfloor$.

To complete the solution, we should be able to calculate function $f(x_1, x_2, y_1) := \sum_{x=x_1}^{x_2-1} \sum_{y=0}^{y_1-1} (x! \cdot y!)$.

Note that $f(x_1, x_2, y_1) = (\sum_{x=x_1}^{x_2-1} (x!) \cdot \sum_{y=0}^{y_1-1} (y!) = (g(x_2) - g(x_1)) \cdot g(y_1)$ for $g(z_0) := \sum_{z=0}^{z_0-1} (z!), 0 \leqslant z \leqslant 10^9$.

Let C be equal to $1\,000\,000$. Precalc and store in the program code two arrays: $fc[i] := (i\ddot{C})!$ and $gc[i] := g(i \cdot C), 0 \leqslant i \leqslant \lfloor \frac{10^9}{C} \rfloor$.

After this step, we can calculate $g(z)$ for any argument $z < m$ in $O(C)$ time; so, our solution contains two constant array with no more than m/C numbers and works in $O(n \cdot C)$.

F. Fill The Boxes

In the computer game “Letters” at the bottom of the screen there are n initially empty boxes. On the top of the screen independently and consequently appear characters, each of them is randomly chosen form the set {A,B,C,D} with respect to some distribution.

As soon as next letter appears, player should place it in one of empty boxes. The game ends when all boxes are filled. Player wins if the letters are arranged lexicographically in non-descending order.

Given the probability for each of letter to appear, calculate the probability for the player to win while playing optimally.

F. Fill The Boxes

Notice that if letter ‘$\widehat{A}$’ appears, the optimal move is to place it into the leftmost empty box.

F. Fill The Boxes

Notice that if letter '$\overset{\frown}{\mathrm{A}}$' appears, the optimal move is to place it into the leftmost empty box.

Similar, if letter '$\overset{\frown}{\mathrm{D}}$' appears, the optimal move is to place it into the rightmost empty box.

F. Fill The Boxes

Notice that if letter 'A' appears, the optimal move is to place it into the leftmost empty box.

Similar, if letter 'D' appears, the optimal move is to place it into the rightmost empty box.

Another idea is that if letter 'B' appears and it has already appeared before during this game, the optimal position is the box near some already placed 'B'. Exactly the same can be said about letter 'C'.

F. Fill The Boxes

Notice that if letter ‘A’ appears, the optimal move is to place it into the leftmost empty box.

Similar, if letter ‘D’ appears, the optimal move is to place it into the rightmost empty box.

Another idea is that if letter ‘B’ appears and it has already appeared before during this game, the optimal position is the box near some already placed ‘B’. Exactly the same can be said about letter ‘C’.

Thus, at any moment of time each letter occupies some consecutive set of boxes.

F. Fill The Boxes

Filling the boxes using the rules above we claim that at every moment of type the game is in one of the following states:

1. Neither ‘$\overset{\frown}{B}$’, nor ‘$\overset{\frown}{C}$’ has appeared and all free boxes form a segment of some length l (like “AAA . . . DD”, “. . . D]”, “. . .]”, “AAA” etc);
2. ‘$\overset{\frown}{B}$’ already appeared at least once, while ‘$\overset{\frown}{C}$’ did not. Free boxes form two segments of lengths ab (to the left of ‘$\overset{\frown}{B}$’-segment) and bd (to the right of the ‘$\overset{\frown}{B}$’-segment)(like “A . . . BBB . . . DD”, “. . . BBB . . . ”, “ABBB”, etc.);
3. ‘$\overset{\frown}{C}$’ appeared, while ‘$\overset{\frown}{B}$’ did not. Again, all free boxes form two segments of lengths ac (to the left of ‘$\overset{\frown}{C}$’-segment) and cd (to the right of the ‘$\overset{\frown}{C}$’-segment); (like “A . . . CCC . . . DD”, “. . . CCC . . . ”, “CCDDD”, etc.)
4. Both ‘$\overset{\frown}{B}$’ and ‘$\overset{\frown}{C}$’ appeared at least once. Free boxes form three segments of length ab (to the left of ‘$\overset{\frown}{B}$’-segment), bc

F. Fill The Boxes

For every possible state of the game we would like to compute the probabilities of winning in case the game is already in this state. For eeach of the four types listed above we would like to maintain a separate array: $d1[l]$, $d2[ab][bd]$, $d3[ac][cd]$, $d4[ab][bc][cd]$. This dynamic programming values can recomputed through each other. For example, consider $d4[ab][bc][cd]$:

- With probability p_1, ‘A’ appears, and ab will be decreased by one.
- With probability p_2, ‘B’ appears, and we are choosing ab or bc to be decreased by one.
- With probability p_3, ‘C’ appears, and we are choosing bc or cd to be decreased by one.
- With probability p_4, ‘D’ appears, and cd will be decreased by one.

F. Fill The Boxes

For similar reasons:

$d3[ac][cd] = p_1 \cdot d3[ac-1][cd] + p_2 \cdot \max_{i=1}^{ac}(d4[i-1][ac-i][cd]) + p_3 \cdot max(d3[ac-1][cd], d3[ac][cd-1]) + p_4 \cdot d3[ac][cd-1];$

similar formula can be written for d_2 and d_1.

F. Fill The Boxes

For similar reasons:

$d3[ac][cd] = p_1 \cdot d3[ac-1][cd] + p_2 \cdot \max_{i=1}^{ac}(d4[i-1][ac-i][cd]) +$
$p_3 \cdot max(d3[ac-1][cd], d3[ac][cd-1]) + p_4 \cdot d3[ac][cd-1]$;
similar formula can be written for d_2 and d_1.

The answer is $d1[n]$; complexity of this solution is $O(n^3)$.

G. Grid With Honey

Given set of hexagonal cells; some of these cells contain honey, and others do not. At the end of the day cell contains honey if and only if the number of neighbouring cells that contained honey at the end of the previous day was odd. The task is to find the placement of honey in honeycomb at the end of day k.

G. Grid With Honey

Enumerate all cells from 1 to $n \cdot m$. Let $v_k = (v_{k_1}, v_{k_2}, \ldots, v_{k_{nm}})$ be a binary vector such that $v_{k_j} = 1$ if on the k-th day, j-th cell contains honey.

G. Grid With Honey

Enumerate all cells from 1 to $n \cdot m$. Let $v_k = (v_{k_1}, v_{k_2}, \ldots, v_{k_{nm}})$ be a binary vector such that $v_{k_j} = 1$ if on the k-th day, j-th cell contains honey.

Let A be $(nm) \times (nm)$ binary matrix, such that $A_{ij} = 1$ if i-th and j-th cells are neighbouring.

G. Grid With Honey

Enumerate all cells from 1 to $n \cdot m$. Let $v_k = (v_{k_1}, v_{k_2}, \ldots, v_{k_{nm}})$ be a binary vector such that $v_{k_j} = 1$ if on the k-th day, j-th cell contains honey.

Let A be $(nm) \times (nm)$ binary matrix, such that $A_{ij} = 1$ if i-th and j-th cells are neighbouring.

Then, for any non-negative integer l: $v_{l+1} = v_l \cdot A \ mod\ 2$, so $v_k = v_0 \cdot A^k$ and can be calculated in $O(n^3 \cdot \log k)$ using binary exponentation.

H. How To Cheat In Lottery

Given undirected graph consisting of n vertices, so that for any two vertices u and v, there is no triple of paths connecting u and v, such that any two of them share no common vertices except v_1 and v_2.

For each $l = 0, 1, \ldots, n-1$ we have to find the number of simple paths of length l modulo $m = 10^9 + 7$.

H. How To Cheat In Lottery

Without loss of generality we can only consider the case of connected graph. One can show that the above definition means that the given graph is cactus, i.e. connected graph, s.t. each edge belongs to no more than one simple cycle. We will assume that any bridge in our graph is a cycle of length 2.

H. How To Cheat In Lottery

Without loss of generality we can only consider the case of connected graph. One can show that the above definition means that the given graph is cactus, i.e. connected graph, s.t. each edge belongs to no more than one simple cycle. We will assume that any bridge in our graph is a cycle of length 2.

With a single depth-first search run we can find all cycles in the cactus; any cycle will be formed by some path in DFS-tree and one back edge (edge not used in DFS-tree). Write these cycles in vector *cycles* (in fact, vector of vectors), and for each vertex keep track on the index of cycles this vertex lies in (and its position in the cycle).

H. How To Cheat In Lottery

Then, we will use function *solve* which works in a following way:

H. How To Cheat In Lottery

Then, we will use function *solve* which works in a following way:

1. *solve* will be similar to *dfs* on subtrees (just the same, but on cactus);

H. How To Cheat In Lottery

Then, we will use function *solve* which works in a following way:

1. *solve* will be similar to *dfs* on subtrees (just the same, but on cactus);
2. arguments of *solve* is some vertex v and parent cycle of it's vertex (or -1 if there are no parent cycle);

H. How To Cheat In Lottery

Then, we will use function *solve* which works in a following way:

1. *solve* will be similar to *dfs* on subtrees (just the same, but on cactus);
2. arguments of *solve* is some vertex v and parent cycle of it's vertex (or -1 if there are no parent cycle);
3. $solve(v, parentCycleNumber)$ will return a vector d such that $d[i]$ will be number of paths of length i starting in v which don't have any intersections with parent cycle except vertex v;

H. How To Cheat In Lottery

Then, we will use function *solve* which works in a following way:

1. *solve* will be similar to *dfs* on subtrees (just the same, but on cactus);
2. arguments of *solve* is some vertex v and parent cycle of it's vertex (or -1 if there are no parent cycle);
3. $solve(v, parentCycleNumber)$ will return a vector d such that $d[i]$ will be number of paths of length i starting in v which don't have any intersections with parent cycle except vertex v;
4. answer vector d will not have ending zeroes; it means that size of d will be no more than "number of vertices in subtree";

H. How To Cheat In Lottery

Then, we will use function *solve* which works in a following way:

1. *solve* will be similar to *dfs* on subtrees (just the same, but on cactus);
2. arguments of *solve* is some vertex v and parent cycle of it's vertex (or -1 if there are no parent cycle);
3. $solve(v, parentCycleNumber)$ will return a vector d such that $d[i]$ will be number of paths of length i starting in v which don't have any intersections with parent cycle except vertex v;
4. answer vector d will not have ending zeroes; it means that size of d will be no more than "number of vertices in subtree";
5. *solve* will work using memorisation;

H. How To Cheat In Lottery

Then, we will use function *solve* which works in a following way:

1. *solve* will be similar to *dfs* on subtrees (just the same, but on cactus);
2. arguments of *solve* is some vertex v and parent cycle of it's vertex (or -1 if there are no parent cycle);
3. $solve(v, parentCycleNumber)$ will return a vector d such that $d[i]$ will be number of paths of length i starting in v which don't have any intersections with parent cycle except vertex v;
4. answer vector d will not have ending zeroes; it means that size of d will be no more than "number of vertices in subtree";
5. *solve* will work using memorisation;
6. $solve(v, parentCycleNumber)$ works with all cycles except parent one; for each cycle c, *solve* runs recursively from each vertex u of c except v and update answer for v with $solve(u, c)$ and two "distances" (on the cycle) from u to v.

H. How To Cheat In Lottery

H. How To Cheat In Lottery

Non-recursive part each $solve(v, parentCycleNumber)$ works in $O(n)$ time, because summary number of vertices in "subtrees" is no more than n.

H. How To Cheat In Lottery

Non-recursive part each $solve(v, parentCycleNumber)$ works in $O(n)$ time, because summary number of vertices in "subtrees" is no more than n.

To find answer, we sum up result of $solve(v, -1)$ for all vertices v; then number of pairs $v, parentCycleNumber)$ we calculate $solve$ for is $O(n)$, and algorithm works in $O(n^2)$ time.

H. How To Cheat In Lottery

Non-recursive part each $solve(v, parentCycleNumber)$ works in $O(n)$ time, because summary number of vertices in "subtrees" is no more than n.

To find answer, we sum up result of $solve(v, -1)$ for all vertices v; then number of pairs $v, parentCycleNumber)$ we calculate $solve$ for is $O(n)$, and algorithm works in $O(n^2)$ time.

Next slide contains main part of author's implementation of code.

H. How To Cheat In Lottery

```
void update(vector<int>& a, vector<int>& b, int d) {
  if (b.size() + d > a.size()) a.resize(b.size() + d, 0);
  for (int i = 0; i < (int)b.size(); ++i)
    a[i + d] = add(a[i + d], b[i]);
}

map<pair<int, int>, vector<int>> memo;

vector<int>& solve(int x, int dad) {
  if (memo.count({x, dad})) return memo[{x, dad}];

  memo[{x, dad}] = {0, 1};
  auto& ans = memo[{x, dad}];

  for (auto p: c[x]) {
    int i = p.first;
    int pos = p.second;
    int c_len = cycles[i].size();
    if (i == dad) continue;

    for (int j = 0; j < c_len; ++j) {
      int d = (j - pos + c_len) % c_len;
      int y = cycles[i][j];
      if (d == 0) continue;

      update(ans, solve(y, i), d);
      if (c_len > 2) update(ans, solve(y, i), c_len - d);
    }
  }
  return ans;
}
```

I. Inside The Matrix

Given an $n \times n$ array filled by numbers from 1 to n^2, we are to find a sum of elements in rectangular subarray.

I. Inside The Matrix

Suppose for simplicity then n is even (odd case is simlilar). Then, the first idea is to use principle of inclusion-exclusion and reduce a problem to a case $r_1 = c_1 = 1$; this case is reduced to four cases:

1. $1 = r_1 \leqslant r_2 \leqslant n/2, 1 = e_1 \leqslant e_2 \leqslant n/2;$
2. $1 = r_1 \leqslant r_2 \leqslant n/2, n/2 + 1 = e_1 \leqslant e_2 \leqslant n;$
3. $n/2 + 1 = r_1 \leqslant r_2 \leqslant n, 1 = e_1 \leqslant e_2 \leqslant n/2;$
4. $n/2 + 1 = r_1 \leqslant r_2 \leqslant n, n/2 + 1 = e_1 \leqslant e_2 \leqslant n.$

I. Inside The Matrix

Consider a first type of problem; let R be given rectangle. Divide cells of given rectangle to two "ladders"
$S1 = \{(x, y) \in R | (x \geqslant y)\}$ and $S2 = \{(x, y) \in R | (x < y)\}$.

It's obvious that for each ladder, sum of the value of our ladder is the sum of values of arithmetic progressions with step 1, whose lengths are an arithmetic progression with step 1 or -1, and and sequence of the first values is such that differences between neigbouring values form an arithmetic progression with step 4 or -4.

It can be proved that the sum of each such "ladder" is a polynomial function of r_2 and e_2, and degree of the polynom is no more than 4. This polynomial can be found as explicit formula or interpolated using some number of first elements of the "ladder".

I. Inside The Matrix

In each of subproblem, the rectangle can be splitted into some number of similar sequences of arithmetic progressions in a similar way, and answer can be obtained.

The complexity of the solution is $O(1)$; to avoid a problem with dividing by non-prime modulo, it's recommended to use `__int128` in C++ and `BigInteger` in Java.

J. Join Them With Or

Given a sequence $A = (a_1, a_2, \ldots, a_n)$ of 31-bit non-negative integers; for every integer k between 1 and n find x_k equal to maximum possible bitwise OR of k consecutive elements of A.

J. Join Them With Or

Designate *bitLength* be maximum possible bits in a_i-s (31 in given constrains), and $OR[l, r] := a_l OR a_{l+1} OR \ldots OR a_r$.

J. Join Them With Or

Designate *bitLength* be maximum possible bits in a_i-s (31 in given constrains), and $OR[l, r] := a_l OR a_{l+1} OR \dots OR a_r$.

Also, let $d[i][j], 1 \leqslant i \leqslant n, 0 \leqslant j \leqslant bitLength - 1$ be $max\{k | k \leqslant i$, and j-th bit of number a_k is $1\}$ or 1 if such k does not exist. All values of $d[i][j]$ can be easily calculated in $O(n \cdot bitLength)$ ($d[i][j] = i$ if j-th bit of i is 1 and $d[i-1][j]$ otherwise).

J. Join Them With Or

Designate $bitLength$ be maximum possible bits in a_i-s (31 in given constrains), and $OR[l, r] := a_l OR a_{l+1} OR \dots OR a_r$.

Also, let $d[i][j], 1 \leqslant i \leqslant n, 0 \leqslant j \leqslant bitLength - 1$ be $max\{k | k \leqslant i$, and j-th bit of number a_k is $1\}$ or 1 if such k does not exist. All values of $d[i][j]$ can be easily calculated in $O(n \cdot bitLength)$ ($d[i][j] = i$ if j-th bit of i is 1 and $d[i-1][j]$ otherwise).

Also let $dor[i][j], 1 \leqslant i \leqslant n, 0 \leqslant j \leqslant bitLength - 1$, is equal to $OR[d[i][j], i]$ (this value can be calculated in similar way).

The key idea of the solution is that for all $1 < l \leqslant r \leqslant n$:

1. $OR[l, r] \leqslant OR[l-1, r]$;

J. Join Them With Or

Designate $bitLength$ be maximum possible bits in a_i-s (31 in given constrains), and $OR[l, r] := a_l OR a_{l+1} OR \dots OR a_r$.

Also, let $d[i][j], 1 \leqslant i \leqslant n, 0 \leqslant j \leqslant bitLength - 1$ be $max\{k | k \leqslant i$, and j-th bit of number a_k is $1\}$ or 1 if such k does not exist. All values of $d[i][j]$ can be easily calculated in $O(n \cdot bitLength)$ ($d[i][j] = i$ if j-th bit of i is 1 and $d[i-1][j]$ otherwise).

Also let $dor[i][j], 1 \leqslant i \leqslant n, 0 \leqslant j \leqslant bitLength - 1$, is equal to $OR[d[i][j], i]$ (this value can be calculated in similar way).

The key idea of the solution is that for all $1 < l \leqslant r \leqslant n$:

1. $OR[l, r] \leqslant OR[l-1, r]$;
2. $OR[l, r] < OR[l-1, r]$ if and only if $l - 1 = d[r][j]$ for some $j \in [0 \dots bitLength - 1]$.

J. Join Them With Or

Designate $bitLength$ be maximum possible bits in a_i-s (31 in given constrains), and $OR[l, r] := a_l OR a_{l+1} OR \dots OR a_r$.

Also, let $d[i][j], 1 \leqslant i \leqslant n, 0 \leqslant j \leqslant bitLength - 1$ be $max\{k | k \leqslant i$, and j-th bit of number a_k is $1\}$ or 1 if such k does not exist. All values of $d[i][j]$ can be easily calculated in $O(n \cdot bitLength)$ ($d[i][j] = i$ if j-th bit of i is 1 and $d[i-1][j]$ otherwise).

Also let $dor[i][j], 1 \leqslant i \leqslant n, 0 \leqslant j \leqslant bitLength - 1$, is equal to $OR[d[i][j], i]$ (this value can be calculated in similar way).

The key idea of the solution is that for all $1 < l \leqslant r \leqslant n$:

1. $OR[l, r] \leqslant OR[l-1, r]$;
2. $OR[l, r] < OR[l-1, r]$ if and only if $l - 1 = d[r][j]$ for some $j \in [0 \dots bitLength - 1]$.
3. $OR[l, r] = max(a_r, max\{dor[r][j] | d[r][j] \geqslant l\})$ (consequence of statements 1 and 2).

J. Join Them With Or

Designate $bitLength$ be maximum possible bits in a_i-s (31 in given constrains), and $OR[l, r] := a_l OR a_{l+1} OR \ldots OR a_r$.

Also, let $d[i][j], 1 \leqslant i \leqslant n, 0 \leqslant j \leqslant bitLength - 1$ be $max\{k | k \leqslant i$, and j-th bit of number a_k is $1\}$ or 1 if such k does not exist. All values of $d[i][j]$ can be easily calculated in $O(n \cdot bitLength)$ ($d[i][j] = i$ if j-th bit of i is 1 and $d[i-1][j]$ otherwise).

Also let $dor[i][j], 1 \leqslant i \leqslant n, 0 \leqslant j \leqslant bitLength - 1$, is equal to $OR[d[i][j], i]$ (this value can be calculated in similar way).

The key idea of the solution is that for all $1 < l \leqslant r \leqslant n$:

1. $OR[l, r] \leqslant OR[l-1, r]$;
2. $OR[l, r] < OR[l-1, r]$ if and only if $l - 1 = d[r][j]$ for some $j \in [0 \ldots bitLength - 1]$.
3. $OR[l, r] = max(a_r, max\{dor[r][j] | d[r][j] \geqslant l\})$ (consequence of statements 1 and 2).

J. Join Them With Or

Let Y_k be $max\{dor[r][j] | 1 \leqslant r \leqslant n, d[r][j] = r - k + 1\}$; array of y_k-s can be simply found from $d[i][j]$ and $dor[i][j]$;

J. Join Them With Or

Let Y_k be $max\{dor[r][j] | 1 \leqslant r \leqslant n, d[r][j] = r - k + 1\}$; array of y_k-s can be simply found from $d[i][j]$ and $dor[i][j]$;

then $x_k = OR(y_1, y_2, ..., y_k)$; all x_k-s can be calculated in $O(n)$.

J. Join Them With Or

Let Y_k be $max\{dor[r][j] | 1 \leqslant r \leqslant n, d[r][j] = r - k + 1\}$; array of y_k-s can be simply found from $d[i][j]$ and $dor[i][j]$;

then $x_k = OR(y_1, y_2, ..., y_k)$; all x_k-s can be calculated in $O(n)$.

The overall complexity of the solution is $O(n \cdot bitLength)$.

K. Key Validation

Given a sequence of sixteen decimal digits, check whether it is valid according to Luhn's formula.

K. Key Validation

In this problem, one has to implement the algorithm described in the statements :)

L. Letter Manipulations

Given the string s, consisting of lowercase English letters and integer k. We are to replace no more than k letters by some other letters such as length of the longest substring, appearing in the new string at least twice, is maximized.

L. Letter Manipulations

Let n be a length of string s, and $s[l \dots r]$ be substring of string s, containing characters from l-th to r-th inclusive; $s = s[1 \dots n]$.

L. Letter Manipulations

Let n be a length of string s, and $s[l \dots r]$ be substring of string s, containing characters from l-th to r-th inclusive; $s = s[1 \dots n]$.

The idea is to compute for all pairs of $diff$ and l, $1 \leqslant diff \leqslant n - 1$, $l + diff \leqslant n$ value $d[diff][l]$ — maximum integer number len, $l + diff + len - 1 \leqslant n$, such that $s[l \dots l + len - 1]$ and $s[l + diff \dots l + diff + len - 1]$ may be made equal by changing no more than k letters. Obviously, the answer can be computed in $O(n^2)$ if we have all d-s calculated.

L. Letter Manipulations

Let n be a length of string s, and $s[l \dots r]$ be substring of string s, containing characters from l-th to r-th inclusive; $s = s[1 \dots n]$.

The idea is to compute for all pairs of $diff$ and l, $1 \leqslant diff \leqslant n - 1$, $l + diff \leqslant n$ value $d[diff][l]$ — maximum integer number len, $l + diff + len - 1 \leqslant n$, such that $s[l \dots l + len - 1]$ and $s[l + diff \dots l + diff + len - 1]$ may be made equal by changing no more than k letters. Obviously, the answer can be computed in $O(n^2)$ if we have all d-s calculated.

Note that $s[diff][l] \leqslant s[diff][l + 1] + 1$ for any "good" pair $diff$, l. If some value of $diff$ is fixed, $s[diff][l]$ can be computed for all l from $n - diff$ to 1 (in reverse order) using so called "two pointers" optimization technique. Let r be "a second pointer". Then, we want to be able to perform the following operations:

L. Letter Manipulations

Let n be a length of string s, and $s[l \dots r]$ be substring of string s, containing characters from l-th to r-th inclusive; $s = s[1 \dots n]$.

The idea is to compute for all pairs of $diff$ and l, $1 \leqslant diff \leqslant n-1$, $l + diff \leqslant n$ value $d[diff][l]$ — maximum integer number len, $l + diff + len - 1 \leqslant n$, such that $s[l \dots l + len - 1]$ and $s[l + diff \dots l + diff + len - 1]$ may be made equal by changing no more than k letters. Obviously, the answer can be computed in $O(n^2)$ if we have all d-s calculated.

Note that $s[diff][l] \leqslant s[diff][l+1] + 1$ for any "good" pair $diff$, l. If some value of $diff$ is fixed, $s[diff][l]$ can be computed for all l from $n - diff$ to 1 (in reverse order) using so called "two pointers" optimization technique. Let r be "a second pointer". Then, we want to be able to perform the following operations:

1. decrease l;
2. decrease r;

L. Letter Manipulations

To maintain this value, observe that $s[l \dots r]$ and $S[l + \mathit{diff} \dots r + \mathit{diff}]$ are equal if and only if $\forall \mathit{rem} \in \{0, 1, \dots, \mathit{diff} - 1\} \forall x, y \in [l \dots r] \cup [l + \mathit{diff} \dots r + \mathit{diff}], (y - x) \ \mathit{mod} \ \mathit{diff} = 0 : s[x] = s[y]$.

L. Letter Manipulations

To maintain this value, observe that $s[l \dots r]$ and $S[l + \mathit{diff} \dots r + \mathit{diff}]$ are equal if and only if $\forall \mathit{rem} \in \{0, 1, \dots, \mathit{diff} - 1\} \forall x, y \in [l \dots r] \cup [l + \mathit{diff} \dots r + \mathit{diff}], (y - x) \ \mathit{mod} \ \mathit{diff} = 0 : s[x] = s[y]$.

Let $\mathit{num}[\mathit{rem}][\mathit{letter}], \mathit{rem} \in \{0, 1, \dots, \mathit{diff} - 1\}, \mathit{letter} \in \{\text{'a'}, \text{'b'}, \dots, \text{'z'}\}$ be a number of positions pos such that $\mathit{pos} \in [l \dots r] \cup [l + \mathit{diff}, r + \mathit{diff}]$, $\mathit{pos} \mod \mathit{diff} = \mathit{rem}$ and $s[\mathit{pos}] = \mathit{letter}$.

L. Letter Manipulations

To maintain this value, observe that $s[l \dots r]$ and $S[l + \mathit{diff} \dots r + \mathit{diff}]$ are equal if and only if $\forall rem \in \{0, 1, \dots, \mathit{diff} - 1\} \forall x, y \in [l \dots r] \cup [l + \mathit{diff} \dots r + \mathit{diff}], (y - x)\ mod\ \mathit{diff} = 0 : s[x] = s[y]$.

Let $num[rem][letter], rem \in \{0, 1, \dots, \mathit{diff} - 1\}, letter \in \{\text{'a'}, \text{'b'}, \dots, \text{'z'}\}$ be a number of positions pos such that $pos \in [l \dots r] \cup [l + \mathit{diff}, r + \mathit{diff}]$, $pos \mod \mathit{diff} = rem$ and $s[pos] = letter$.

Also let $numMax[rem], rem \in \{0, 1, \dots, \mathit{diff} - 1\}$ be $\max\limits_{letter \in \{\text{'a'}, \text{'b'}, \dots, \text{'z'}\}} num[rem][letter]$.

L. Letter Manipulations

To maintain this value, observe that $s[l \dots r]$ and $S[l + diff \dots r + diff]$ are equal if and only if $\forall rem \in \{0, 1, \dots, diff - 1\} \forall x, y \in [l \dots r] \cup [l + diff \dots r + diff], (y - x) \ mod \ diff = 0 : s[x] = s[y]$.

Let $num[rem][letter]$, $rem \in \{0, 1, \dots, diff - 1\}$, $letter \in \{\text{'a'}, \text{'b'}, \dots, \text{'z'}\}$ be a number of positions pos such that $pos \in [l \dots r] \cup [l + diff, r + diff]$, $pos \mod diff = rem$ and $s[pos] = letter$.

Also let $numMax[rem]$, $rem \in \{0, 1, \dots, diff - 1\}$ be $\max\limits_{letter \in \{\text{'a'}, \text{'b'}, \dots, \text{'z'}\}} num[rem][letter]$.

Then, a minimal number we want to know is $r - l + 1 - \sum\limits_{rem=0}^{diff-1} numMax[rem]$; let's contain and support it in variable $symbolsToReplace$.

L. Letter Manipulations

There are many ways to support *num*, *numMax* and *symbolsToReplace* so that when l or r decrease by one, it's possible to modify these arrays and variables in $O(1)$ time (instead of $O(|\Sigma|)$); one of the ways is to maintain an additional array $countNum[rem][num]$ ("how many $num[rem][letter]$ with fixed rem are equal to num"). Details of the implementation are left to the reader as an excercise.

L. Letter Manipulations

There are many ways to support *num*, *numMax* and *symbolsToReplace* so that when l or r decrease by one, it's possible to modify these arrays and variables in $O(1)$ time (instead of $O(|\Sigma|)$); one of the ways is to maintain an additional array $countNum[rem][num]$ ("how many $num[rem][letter]$ with fixed rem are equal to num"). Details of the implementation are left to the reader as an excercise.

The running time for fixed value of *diff* is linear ($O(n)$), thus the overall running time is $O(n^2)$.